Usable Online Privacy:
Browser-Based End-to-End Encryption Delivered by a Service Provider

Joshua Korner-Godsiff - u4685222
Supervised by Roger Clarke

16th September 2014

This page unintentionally left blank.

Except for this bit. This bit was intentional.

Abstract

Internet users are facing increasingly far-reaching and sophisticated threats to their privacy, from a broad range of sources. Such users are storing more and more of their data on other people's servers, and anyone who gains access to these servers can obtain all the data stored there. I will detail these threats, and argue that there exists a need for a method of protecting against them. Moreover, I will argue that this method needs to be usable by the vast majority of internet users if any real progress is to be made in terms of protecting the privacy of the general public.

I will then present a model by which organisations that supply services over the internet can protect the privacy of their users, with minimal user involvement necessary in this process. In particular, this model will ensure that even if someone gains access to the servers of these organisations, they cannot obtain access to the data stored there.

I will then present an additional model, under which users can verify that their privacy is in fact being protected under the first model.

Contents

Chapter 1

Introduction

In the course of using a web-based service, users will often disclose an enormous amount of personal and private information to that service, without necessarily realising where that information may ultimately end up, or how it may be used. Despite the best assurances of these services, their users often have no guarantee - legal, technical, or otherwise - that their data will not end up in the hands of some third party. Such third parties can include anyone from advertising firms, to hackers, to government agencies[1]. The means by which they may acquire such data is similarly varied, from being willingly handed it by the service operator, to acquiring it illegally by compromising the security of the server, to compelling the service operator to disclose it via legal means. As a result of that final avenue in particular, even if a service is committed to protecting the privacy of its users, and takes all reasonable security precautions, it logically cannot guarantee its users privacy so long as it itself has the ability to access the data. In the case of services which are *not* committed to protecting their users, there are even further threats: an unscrupulous service provider can hold data to ransom or commit extortion, and on the more benign side one could quite deftly manipulate a user's behaviour. Google is quietly becoming remarkably good at the latter - an Android phone can now tell you which roads to drive down, where to eat, and what stocks to watch, among many other things.

Concern over these potential intrusions has been greatly heightened within the previous several years. The leaks from Edward Snowden [14] have revealed that the United States' National Security Agency (NSA), and Britain's Government Communications Headquarters (GCHQ) have been engaging in almost carte blanche surveillance of internet activity. They reportedly have access to data related to users of Facebook, Google, Microsoft, Apple, Yahoo, and a number of other companies [14]. While most such companies have stated that they only turn over data in response to a legally binding order or subpoena, it has been alleged that the access by intelligence agencies to their data is far more streamlined and direct.

Many other nations' intelligence agencies - notably that of Canada, Australia, and New Zealand, which together with the US and Britain makeup the so-called "Five Eyes" alliance, but also including that of Austria, India, Netherlands, Russia, China, Iran, Syria, and Israel - are all either known or alleged to be participating in such activity to varying degrees.

Of further concern is the reality that companies and other organisations can be compelled by a court to turn over their encryption keys [13, 36] to a government, an action which undermines one of the cornerstones upon which internet security is based. Not only does the possessing of such encryption keys provide access to user data, but it also enables a third party to imper-

[1] Not necessarily the agencies of the user's own government, either.

sonate the service, effectively tricking users into believing they are using the real thing, when in fact they are using a decoy designed for surveillance.

Threats to privacy and security from criminal elements also cannot be taken lightly. In 2013, a hack of the United States branch of the Target corporation resulted in around forty million credit-card records, and seventy million records of addresses, telephone numbers and other instances of personal information ending up in the hands of hackers. Moreover, this happened in the face of a security system which allegedly detected the security breach as it was in progress, but failed to take any action to prevent it [26]. In 2011, hackers breached Sony's PlayStation Network, making off with around seventy million customer data records, including names addresses, dates of birth, hashed and salted passwords, and security questions [45], forcing the network offline for twenty-four days. These hacks are alleged to have cost their respective companies millions of dollars, and enabled waves of identity and credit-card fraud, and other crimes, and they are far from the only examples of such crimes.

In the face of such threats to both privacy and security, the questions must be asked: What can the average user do to protect themselves against such threats? And what can the average company do to protect both themselves and their users?

A potential solution lies in the idea of encrypting all sensitive information prior to transmission (for example, in the user's web-browser), and only storing the encrypted data on the server[2]. Under such a model, even if the server or the service operator is completely compromised, the private data of the users would remain private, as whomever had compromised the server would lack access to the necessary decryption keys. Such a model has already been adopted (at least in some form) by a number of web-based services; examples include Mega.co.nz [23], CryptoCat [20] and ProtonMail [53], all of which offer variants on the ideas that will be presented in this thesis.

The idea has the potential to neatly deal with all three primary sources of threat to privacy (hackers, advertisers, and government agencies), and for the last of those it is all but necessary. Server-side encryption, combined with proper security procedures, is generally sufficient to mitigate a technical threat from hackers - at least to the point where the difficulties of client-side encryption are not worth the security gain. Legalistic means (either via legislation, or via the service's privacy policy) have the potential to remove the possibility of data being handed to third parties, although many nations lack sufficient protections to make this a reality, and privacy policies may not be sufficiently protective (or enforceable).

Governments, on the other hand, can exercise considerable power and influence over corporations and other organisations under their jurisdiction[3], forcing them to act in ways they otherwise would not, including circumventing their own security measures. The government $\leftrightarrow$ corporation power dynamic is not even the only one to have these concerns - merely the most obvious one. In certain parts of the world, such as Russia, South-East Asia, and Central/Latin America, it is quite conceivable that organised crime syndicates could exercise a similar degree of leverage over internet companies, and other organisations with large amounts of sensitive data.

In this paper, the fundamentals of a model for performing client-side encryption of user-data for web services will be presented. The naive version of such a model suffers from a number

[2] That is, do not store passwords, encryption keys, or plain-text data on the server.

[3] I should probably note, at this point, that I don't actually object to governments being *able* to exercise such power. I object to the what they specifically doing with it, and particularly the *scope* of what they're doing with it.

It must also be noted that we are not exclusively dealing with western democratic governments, who at worst tend to lock people up. In less democratic parts of the world, governments having mass-surveillance capabilities tends to get people killed.

of serious security and usability concerns which (if unsolved) undermine its core purpose; we shall therefore address as many of these concerns as possible, and make note of the ones which we cannot.

The first concern is that - at least in the case of an HTML+JavaScript based web-service - a compromised service could provide malicious code to the client, enabling it to retrieve sensitive data and/or decryption keys, completely undermining the client-side security mechanisms. Solving this in the case of web-applications is particularly difficult due to the highly dynamic nature of web-pages and JavaScript.

Secondly, it is unreasonable to expect that all users of a service would be willing, able, know that solutions exist, and care enough to install software which ensures protection of their privacy. Indeed, possibly the most problematic aspect of many existing privacy-oriented pieces of software (a prime example being PGP [2]) is that they set an unfortunately high bar in terms of the required skills and knowledge in order to use them effectively, while simultaneously doing little to promote their use to the general public. Any solution which aims to to provide protection against mass surveillance and mass data-theft must provide its protection to as many users as possible, and thus must necessarily minimise the barriers to uptake and use.

Thirdly, many modern web-applications provide the ability for users to share data with one another. If such data is encrypted, then the encryption keys must be distributed to all relevant users in order allow for sharing. If the key exchange and distribution is handled by the server, then a compromised server could either decrypt the files en-route (in the case of symmetric encryption) or substitute keys belonging to users with ones they control (in the case of asymmetric encryption), and neither of these cases would be easily detectable by the end-users.

Fourthly, users must be able to have some certainty that their privacy, and the security of their data is being preserved, beyond even the assurances provided to them by the providers of the services they use, or the governments of the appropriate jurisdictions.

This paper shall address all of these problems in detail. Additionally, we shall make careful note of (though not attempt to solve) a number of other problems that arise, including the issues of password recovery, shifting application logic and processing to the client, and the impact that client-side encryption has on certain popular business models.

1.1 Thesis Structure

Chapter 2 will define the terms used throughout this paper, the threat model we seek to protect against, as well as the goals and requirements of any solution.

Chapter 3 will provide a broad look at the relevant literature and techniques applicable to the the problem, as well an examination of how this problem is being addressed within the web-services industry, as there seems to be more happening on that front than in academia.

Chapter 4 will provide an overview of the solution, along with a discussion of why the particular approach presented in this thesis was chosen.

Chapter 5 will present a model of how to construct a web-service with the intent to protect the privacy of the service's users.

Chapter 6 will present some of the specific implementation details needed. Specifically, those required by the plugin in Chapter 7.

Chapter 7 will present details of a browser-plugin which allows any user who wishes to do so to verify that their privacy is in fact protected by services utilising the model presented in Chapter 5.

Chapter 8 will discuss the strengths and weaknesses of the work presented in this thesis, and suggest promising avenues of further research.

Chapter 2

Problem Definition

2.1 Definition of Terms

We shall firstly define, in as unambiguous a manner as possible, the specific meaning of the terms used throughout this document:

Web-Service A piece of software, or collection of pieces of software, which provide a service to end users, specifically via the world-wide-web.

Web-Service Provider A company or other organisation responsible for the operation of a web-service.

Server The hardware and underlying software stack (OS, web server, database server, etc) upon which the 'server-side' portion of a web-service runs.

User A person who uses a web-service.

Client The hardware and underlying software stack that a user uses to access and interact with a web-service. In applicable cases, includes the browser.

Browser A piece of software capable of interpreting HTML, CSS, and JavaScript, and other relevant data in order to render an holistic web page. For our purposes, it is assumed to run on the client, and is used by a user. Examples include Firefox, Chrome, Internet Explorer, Safari, and Opera.

Sensitive Data Data belonging to and/or originating from the user, and which the user would prefer to be kept private, and/or which could exploited to the user's detriment if it were in the hands of a third party.

Service Critical Data Data without which a web-service is logically unable to perform its intended function. Examples include search terms for a search engine, and the address field in an email.

"Web-Service" and "Web-Service Provider" will sometimes be shortened to simply "Service" and "Service Provider" throughout this text.

2.2 Goal

The overarching goal of this paper is to define a model under which the sensitive data that a user supplies to a web-service *cannot*, as a result of supplying it to the service, fall into the hands of any party to whom the user has not granted access.

It is preferable that this goal be achievable without the user's active participation in achieving it. That is to say, the aim is that all users receive protection, regardless of whether or not they seek it. The reasoning behind this particular constraint is the observation that while there exist a great many services and technologies which allow users to protect their privacy and data, almost all them require active participation on the part of the user to do so - for example by downloading, installing, and/or configuring some specific piece of software - which can present a significant barrier to uptake. This is in contrast to the vast majority of highly successful websites, such as Facebook, Google, YouTube, Amazon, eBay, Wikipedia, Twitter, and indeed most other popular sites [15], where the most significant barrier to uptake is the sign-up form. If online privacy is to become at all ubiquitous, it is imperative that it is not difficult to acquire and use.

Furthermore, we seek to allow users to verify that they are receiving such protections. In the case of those users who do not wish to verify it, or who lack the technical competency to verify it, we rely on the existence of other users who do perform the verification. In essence, we seek to create a digital version of the concept of herd immunity, whereby those users who verify their own protection by extension provide some measure of protection to those who do not, under the assumption that the service or any third party is unable to tell the two apart.

2.3 Threat Model

Here we will define the specific range of threats that we seek to protect against. Many of these threats are interrelated in some way, and therefore the threats cannot be considered in isolation to one another.

2.3.1 Web-Service Provider Compromise

It is assumed to be possible for a third party to either convince or compel the service provider to disclose arbitrary sensitive data that it possesses.

Additionally, it is assumed to be possible for a third party to convince or compel the service provider to alter the operation of the service, in any way, resulting in the web service being compromised. This includes, but is not limited to the disclosure of the service-provider's own encryption keys and security certificates, if any, to the third party.

The consequence of these assumptions is that any sensitive data that the service provider possesses cannot be assumed to be secure, even if the service provider has encrypted it with their own keys.

We do not generally assume that the service provider *itself* is acting against the privacy interests of its users - on the basis that any service provider which sets out to protect its users privacy is unlikely to back-flip on that decision. However, it is still a possibility, and presents identical threats to the above. Because it presents identical threats to the above, it is also solved by any solution to the above.

2.3.2 Web Service Compromise

It is assumed to be possible for a third party to arbitrarily alter the operation of the web service software, either by modifying the code which runs upon the web server, or by modifying the code which is sent to the client. This could be made possible by compromising the service provider, or by compromising the server, or by compromising the transmission medium; however, which of those vectors is used is not relevant.

The consequence of this assumption is that the operation of the service - including any parts of that software which are sent to the client - may not be that which either the service operator or the user expect it to be, and may in some way be malicious.

2.3.3 Server Compromise

It is assumed to be possible for a third party to gain full control of the server(s), and all software operating upon it.

The consequence of this assumption is that the third party may then compromise the web service. It may also acquire any sensitive data stored on the server(s).

2.3.4 Client Compromise

For the purposes of our threat model, it is assumed that the user's own machine is *not* compromised by a third party. Protecting against that possibility is beyond the scope of what this thesis is trying to accomplish; moreover, in the event that the user's machine is compromised, any attempt to protect sensitive data created on that machine is essentially moot, anyway.

Similarly, we do not seek to protect against the actions of other users, or what may occur on the machines of other users. In particular, it is possible that a user might convey sensitive information to another user, who then discloses it.

2.3.5 Transmission Layer Compromise

It is assumed that communications between the client and server may be both intercepted and modified by a third party, including in the case where the connection is encrypted (e.g. by TLS/SSL). This is largely an extension of the possibility that the server/service could be compromised, or the service provider's encryption keys could be discovered.

For sanity's sake, we will not assume that the entire TLS/SSL system is broken[1], since if that is gone, the internet has vastly bigger problems than what we are attempting to address here.

2.3.6 Threat Model Summary

We assume that the server; any software running upon it, including the service itself; and the service provider as an organisation may be compromised. We seek to ensure that even under this eventuality, the confidentiality of sensitive data is not compromised.

[1] Recent bugs in OpenSSL to the contrary.

It must be noted that it is not assumed that the service provider sets out to create a service which intentionally distributes sensitive data, although that is to some extent covered under the threat model. Instead, we assume that the initial intention of the service provider is to create a service which preserves the confidentiality of sensitive data, but that some third party - or indeed, the service provider itself - may attempt to subvert this intention further down the road.

While we seek to provide users with the ability to verify that their privacy and data are being protected, such a guarantee can only be provided to those users willing and able to verify it. Those users who are not willing or able will have to suffice with the assurances of the service, and the fact that those users who do perform such verification have not detected any violations.

In essence, we do not trust in the security of any unencrypted data which leaves the client.

2.4 Requirements

As per the threat model (§2.3), the server and/or the web-service provider is the primary vector through which third parties may acquire sensitive data. Some means of acquiring such data - particularly legal means - cannot be defended against by technical measures if the service has the capability to read the data. Given this, we require a model under which neither the web-service software, nor the server upon which it operates, nor the service provider have the capability to gain access to the content of a user's sensitive data.

Additionally to this, we seek to ensure that compromise of the server, the web-service software, or the operations of the service provider by any third party, in any way, does not jeopardise our ability to guarantee that sensitive data will not end up in the hands of a third party.

An exception is made in the case of service critical data. Even if such data is sensitive, if the user does not supply the data in a readable form to the service, then the service cannot perform its function. We therefore leave protection of such data entirely to the user, in the form of their choice of whether or not to utilise the service.

Insofar as possible, the user should not be *required* to be aware that anything is being done to protect their data; particularly they should not be required to be aware of, nor participate in any of the technicalities of protecting their data, beyond what is normal for a web service (for example, using a password).

Chapter 3

Literature Review

3.1 Previous/other work in the area

While there has certainly been work related to many of the individual aspects presented in this paper, with one exception there appears to be a dearth of academic literature on the topic as a whole.

That exception is a paper entitled "Building web applications on top of encrypted data using Mylar" [35], which contains a *substantial* overlap with the work presented here[1]. In particular, it presents a model "to protect data confidentiality in a wide range of web applications against arbitrary server compromises," which is equivalent to the goals of this paper, and it reaches many of the same conclusions. The paper covers four main areas: an architecture for protecting users' sensitive data, the sharing of sensitive data between users, the verification of application code, and computation over encrypted data. In this paper, we shall cover only the first three[2], and reach markedly different conclusions on the third.

3.1.1 Data Tainting

Data tainting - sometimes also called taint checking - is a well established method of analysing a program to determine which aspects of it are influenced by user input, or other external data sources [40]. Traditionally, it is used as a means to detect vulnerabilities in the program; for example, it can be used to determine if user input (or some derivative thereof) is being fed directly to a SQL database, or as a command to an operating system. However, the same technique may be used to determine if data derived from an HTML DOM is being fed into an AJAX request via JavaScript, which is a use-case we are far more interested in.

To perform taint checking on a piece of software requires four distinct things:

1. The software's source-code (or some other representation we may reason about, such as its byte-code).
2. A set of initially tainted data sources - traditionally, this is anything which may contain unsanitised user input.

[1] It must be noted that the paper was only published in April 2014 (i.e. 4/5ths of the way through this project), leaving insufficient time to significantly diverge the two.

[2] The fourth requiring substantially more mathematical knowledge than I possess.

3. A taint propagation model for the particular programming language(s) that the software is written in, that is used to which determine data paths through the program are (or at least may be) tainted.

4. A set of 'taint-sink' functions - that is, functions for which we do not wish to have tainted data as input(s).

Data tainting in JavaScript (which we are specifically interested in) has a somewhat sketchy history. Versions 3 and 4 of Netscape Navigator provided in-browser support for the concept, albeit for quite specific security purposes. This support was extended to some versions of Internet Explorer and Opera, but the overall feature was deprecated in JavaScript 1.2, and is no longer officially supported in any major browser [50].

There are plenty of more modern attempts at implementing some form of taint analysis for JavaScript [5, 12, 49, 51]. All but one [11] of those encountered while researching this paper used a dynamic analysis approach - that is, they analysed the application as it was running, and only analysed those data paths which were actually executed. This is in contrast to the concept of static analysis, which analyses a program's source code before it is run, and examines all control flow paths. The former is more precise, and carries more computational overhead, while the latter is more thorough, and in principle easier to implement.

3.1.2 Cryptography & Storing Sensitive Data

Cryptography is a well established field of mathematics, dealing chiefly with methods of securing information, such that only those who are authorised to do so may discover the contents of said information. [25]

Briefly, information may be encrypted using a combination of an appropriate mathematical algorithm (e.g. AES [47], Blowfish [43], and RSA[38]), and an encryption key - typically a number large enough that an attacker cannot feasibly try all possible numbers. The information (termed cleartext) and key are used as inputs to the algorithm, in order to produce cyphertext (that is, unreadable gibberish).

In order to turn the cyphertext back into cleartext, the cyphertext is used as input into the decryption algorithm corresponding to the encryption algorithm which was used to generate the cyphertext, along with a decryption key. The decryption function is typically some form of reversal of the operations performed in the encryption algorithm, while the decryption key may either be the same as the encryption key (termed a symmetric key), or another number which is mathematically related to the encryption key (termed an asymmetric key).

In modern use, the encryption and decryption algorithms are typically public knowledge - largely on the basis that by exposing them to many trained mathematicians, any flaws are more likely to be discovered and fixed. The keys on the other hand, should always be kept secret, being made available only to those whom the encrypting party wishes to grant access to the encrypted information. It should be noted that since the keys are numbers, anyone may generate their own keys to use with the algorithm, assuming they know how to generate numbers with the specific properties required. The most common desired properties are

1. That the number is very large - typically 128 bits or more for symmetric keys, and 1024 bits or more for asymmetric keys.

2. That the number is either prime, or has a highly probability of being prime.

3. That the number cannot be easily guessed by a means other than brute-force analysis - this is typically satisfied by choosing a random number.

Encryption is most commonly used to allow the secure transfer of information between two parties, while ensuring that no third party who may have intercepted the information is able to read it. However, it is equally possible to encrypt information, and then safely store it with a potentially untrusted third party for later decryption by the encrypting party. Both of these uses are of interest to us.

3.1.3 Public Key Cryptography

Public key cryptography [10, 25], sometimes also known as asymmetric cryptography, is a particular type of cryptography almost exclusively used for sharing information. It functions by generating not one but two cryptographic keys, typically termed a key-pair. When combined with their associated cryptographic algorithm - for example, RSA [38] - these keys have the property that each may be used to decrypt what they other encrypts. However, they may *not* be used to decrypt what they themselves have encrypted.

The most important result of this (at least for our purposes) is that one can safely distribute or otherwise publish one of these keys (termed the public key), and keep the other key (termed the private key) secret. Other people may then encrypt messages using the public key, and only the holder(s) of the private key may decrypt them. Similarly, any message encrypted using the private key is guaranteed to have come from the holder of the private key - assuming that key is held securely.

3.1.4 End to End Encryption

End-to-end encryption, sometimes also called client-side encryption, is a technique whereby information is encrypted by one party, communicated to a second party who possesses the capability to decrypt it, and at no point between the initial encryption and the final decryption does any third party - whether they are assisting the communication or intercepting it - possess the capability to read the message. More simply, the message may only be read at either end of the transmission, hence the name.

Several points must be made with regard to end-to-end encryption. Firstly, that the communication between the two parties may be asynchronous; that is, the encrypted information may be held for an arbitrary amount of time by an intermediary third party (which is presumably helping to facilitate the communication) before being finally transmitted to the second party; there is no requirement that the first and second party are ever come into direct contact with one another during the communication.

The second point is that the first and second parties may in fact be the *same* party. That is, it is entirely reasonable that one might encrypt some information, send it a third party for storage, and later retrieve it for decryption.

Thirdly, that while the contents of the messages are a secret, the sender and recipient are not, as any intermediaries must know where to send the message in order to send it. There exist methods of anonymising the sender and receiver (e.g. TOR [37]), but those will not be dealt with in this thesis.

End-to-end already encryption enjoys fairly widespread use among security conscious people and organisations. However, one of the overriding themes of most implementations of end-to-end encryption is that the software facilitating it is already deployed to both the sending and receiving parties prior to any encrypted communication taking place, and that these deployments are wholly controlled by the two parties. There are very good reasons for using such a model - pre-deployment and total control enormously simplify a lot of security concerns surrounding end-to-end encryption.

3.1.5 Key Exchange Problem

Many web-services provide functionality whereby information - including potentially sensitive information - can be shared between users[3]. If we wish to hide such data from all the intermediaries between the users - including the server - then it becomes necessary to encrypt the data. The question then becomes: how do we distribute the decryption keys to the user(s) at the other end?

The so-called "key exchange problem" is a well trodden area of cryptography and security. The Diffie-Hellman key-exchange algorithm [10] provides a generally sound method of exchanging keys; however, it suffers from two flaws which are critical to our application:

- It fails to provide a method for one user to authenticate the identity of another.
- It assumes that even if the channel by which keys are exchanged is monitored, it may not be tampered with by a third party (i.e. the channel is not vulnerable to an man-in-the-middle attack).

The second flaw is a particular problem in our case, since the server necessarily facilitates communication between users, and it is assumed to be compromised.

There exists a number of methods which attempt to mitigate this problem. The most prominent is the idea of a centralised public key infrastructure (PKI) [16]. Under a PKI scheme, a central entity known to all users - generally termed the certificate authority (CA) - maintains a binding between public keys and the identifies of the user(s) to which they belong. The public keys of the CA are distributed along with any software that uses it, allowing the software to establish a secure connection between itself and the CA. This in turn allows such software to both look up the public key belonging to a particular identity, and to verify that a particular public key does in fact belong to a particular identity.

The downside of a centralised PKI is that it requires trust in the CA. While this is not necessarily a bad thing - most secure communications on the internet work on this basis - it is still not ideal.

There also exists the concept of a *de*centralised PKI, often termed a "web of trust" [16]. Under such a scheme, the users themselves issue certificates binding their public keys to their identity. These certificates are then signed by other users, based on whether or not they trust the certificate to be valid. Given sufficient users signing each others' keys, one user may establish the identity and public key of another user based on who signs their certificate, and who in turn signs *those* users' certificates, and so forth down the chain until the first user encounters enough users whom they trust to accurately sign certificates.

[3] Facebook and email being examples

Like most decentralised systems, web of trusts do not actually preclude working with a central authority, provided most people in the system trust that authority. The downside to web of trust systems is that they can be considerably more difficult to set up and maintain than a centralised one.

3.1.6 In-Browser Cryptography

Of particular interest is the idea of supplying encryption and decryption functionality within a web-browser, or other similar software such as an HTML-enabled email client. There are many ways in which this could be (and to some degree already has been) accomplished.

Firstly, there is the idea of supplying cryptographic functionality via a specialist library or piece of software, through which all communications are piped. From a purely technical perspective, this is the preferred option. Its chief benefit is that the software is installed on the client, and cannot in any way be meddled with by a third party, except by compromising the client itself. Additionally, such an approach has full access to the underlying operating system, and thus easy access to things such as high-quality sources of entropy, and security supportive hardware. This is the approach that SSL [4] has taken, and it has achieved great success in doing so - though a large part of that success is due to the fact that it is comes packaged with most web-browsers. In order to have similar success, any such approach would likely *also* have to be packaged with the browser, as otherwise it is unlikely to receive widespread uptake, and therefore would not be particularly useful to a service seeking to utilise it.

A second approach is the notion of a browser plugin. This also has the benefit of being installed on the client, but with the drawback that plugins tend to be heavily sandboxed by the browser, thus keeping them away from the underlying operating system. It suffers from the same problem as specialist software, in that it needs broad uptake to be particularly useful, and it is less likely to achieve that if the user must specifically install it. One advantage to using a plugin over specialist software is that a plugin generally has much more direct access to the browser's resources, potentially including the JavaScript runtime and DOM of arbitrary pages [28]. This can greatly assist in providing secure cryptographic operations to the browser.

A third approach is the idea of an on-demand download. Under this approach, code to perform cryptographic operations is loaded onto the client from an external source whenever there is - or may be - a need for it [19]. This has the potential to very neatly sidestep the issues of uptake and usability. However, it suffers from the very serious security issue of having cryptographic code originate from an external source. Additionally, in the specific case of web-browsers, the only way to deliver such code would be via JavaScript, which presents its own litany of security issues.

The third option is actually very promising, and is the one investigated within this thesis. The first two have been tried - to varying degrees of success - but the only solutions to achieve widespread use are the ones which come with the software that uses them, and do not require active user involvement to function. The third option at least potentially has both of those properties, if the security issues can be solved. In any case, given a choice between trying to solve the security issues related to using cryptographic code from an external source in JavaScript, and trying to convince and educate the general public to use encryption, the former is quite probably easier.

3.1.7 Cryptography in JavaScript

Historically, encryption in JavaScript (the de facto scripting language of web-browsers and related software) has been quite problematic, because JavaScript has until quite recently lacked a cryptographically secure random number generator. This meant that developers either had to use JavaScript's `Math.random()` function (which is not cryptographically secure), or use Java applets, or a plugin, or to generate the number elsewhere (e.g. on the server) and send it to the client, all of which have their own associated problems.

JavaScript now possesses the `window.crypto.getRandomValues()` function, which in principle provides cryptographically secure random number generation. It is still "in principle" because of two issues: firstly, the technology (and the associated W3C specification) is still in development. At time of writing the function is supported by all major desktop browsers (Chrome, Firefox, IE, Opera, and Safari) and *some* mobile browsers (Chrome, and Firefox) [32], and it is anticipated that support will continue to spread.

The second issue is that the technology relies on the browser to provide good sources of entropy, and these sources are *not* defined in the specification. Specifically:

> *"Implementations should generate cryptographically random values using well-established cryptographic pseudo-random number generators seeded with high-quality entropy, such as from an operating-system entropy source (e.g., "/dev/urandom"). This specification provides no lower-bound on the information theoretic entropy present in cryptographically random values, but implementations should make a best effort to provide as much entropy as practicable." [7]*

The result is that while browsers *should* provide cryptographically random values, there's no real guarantee that they are doing so. However, it is still the best option available in JavaScript, and notably better than Math.random().

3.1.8 AJAX

AJAX - short for **A**syncronous **J**avascript **A**nd **X**ML, and sometimes also called XHR, short for **X**ML **H**TTP **R**equest - is a well-established technique for performing communications between a JavaScript runtime and a server capable of responding to HTTP requests. It is one of the few ways in which a JavaScript runtime running in a browser can communicate with the outside world. It forms the basis of many modern web applications, due to its ability to exchange data with a server after page load [9].

3.2 Industry Review

Despite persistent searching, there nevertheless seems to exists a dearth of academic literature on some of the topics covered in this paper. It was the activity of the internet-application industry which first led me to investigate the possibility of end-to-end encryption in a browser, and the industry does *not* seem to lack interest or activity in this area - particularly since the Edward Snowden leaks [14]. Mega, CryptoCat, ProtonMail, and a host of enterprises are all attempting to supply a range of private services[4]. Because of this, it seems prudent to touch on the major points of interest that exist on the industrial side, and from less traditional sources.

[4] A simple Google search for "Private <class of service>" will tend to turn these up

MEGA

Mega.co.nz has created a file storage/sharing service which attempts to ensure privacy by using in-browser end-to-end encryption. It encrypts the files it stores using randomly generated symmetric keys. These keys are in turn encrypted with a symmetric master key, which is itself encrypted using a hash derived from the user's password. This allows the encryption keys to be stored on the company's servers without actually disclosing them. [24]

Additionally, each user's account has an RSA key pair associated with it. The private portion is encrypted with the symmetric master key, allowing it to also be stored on the server. The key pair is used to allow users to exchange files, and also for authentication. [23]

Mega identifies three large-scale security vulnerabilities with respect to their service [22].

- A man-in-the-middle attack, involving "issuing a valid duplicate SSL certificate in combination with DNS forging and/or attacks on our BGP routes".
- A third party gaining access to the servers hosting https://mega.co.nz/index.html, and replacing that file with a forged version.
- A third party gaining access to their servers, and creating forged key requests for existing data shares.

They claim that the latter two vulnerabilities are detectable in the client.

Criticisms of JavaScript-based Encryption

Matasano Security offers a number of valid (and not so valid) criticisms of JavaScript based cryptography [44]. Their main concern is what they term the "chicken-egg problem" of a service delivering crypto code to the browser. They argue that if you cannot trust a network to deliver a password, or do not trust a server to keep a user's secrets, then you cannot trust that same server or network to deliver security code.

This is indeed one of the most substantial problems blocking widespread uptake of the idea. However, it may not be insurmountable. The network-delivery problem (as they readily admit) can be solved using TLS/SSL. Trusting the code that is delivered is another matter, and requires finding some way to verify that what is being delivered is both the correct code (i.e. unmodified by some third party), and not itself malicious.

Another substantial criticism they level is the lack of a cryptographically secure RNG in JavaScript. However, this problem has been solved (or at least, is in the process of being solved) by `window.crypto.getRandomValues`() [32].

They also claim that the "view-source transparency is illusory". That is, that the inherently open-source nature of browser-executed JavaScript[5] does not actually help, since even a skilled cryptographer would have difficulty determining if a given piece of JavaScript code is secure - particularly due to JavaScript's malleable runtime environment. While this is partially correct, it is also disingenuous - the benefit of view-source transparency is that a person can read the code, and *know* that it is the same code being executed in the browser - something which is very difficult to impossible with compiled binaries [19]. The result is that analysis and verification of the code - both by automated tools and by third-parties - is possible, particularly if some of the malleability of JavaScript is disallowed (for example, by disallowing the use of the `eval` function).

[5] All browsers that I am aware of provide a tool to view the source-code.

Man In The Middle Attack

In any situation where users may exchange encrypted sensitive data using a service, there is the potential for a man-in-the-middle attack to be executed by that service (or by some third party who as gained access to their servers) [19]. Because the server is likely to act as a de facto public key infrastructure - in that it facilitates the exchange of public keys between users - it has the capability to provide a public key for which it controls the private key to the user sending information, and decrypt the information for itself before forwarding it on to the receiving user.

In order to negate this possibility, it is necessary for there to be some way to verify that a particular public key belongs to the user that the service claims it belongs to. The most common methods of doing this are to either pre-distribute the keys to end users, or to use a side channel to facilitate key exchange, or to verify the authenticity of keys via some side channel.

LavaBit, and the Assumed Security of TLS/SSL

Most discussions of internet- and web-centric security assume that TLS/SSL is as secure as it reasonably can be. However, there is reason to believe that this is not necessarily the case.

LavaBit.com [21] was an email provider (it has since shut down) specialising in providing privacy to its customers. It did so by encrypting emails "at rest" (i.e. after it received them). The mechanisms it used to do this meant that it still retained some ability to access those emails at certain points in the chain, although it actively chose not to do so.

Recently unsealed court documents show that the FBI demanded (via the court) that LavaBit.com hand over its SSL keys [13, 36], presumably in an attempt to monitor or pursue Edward Snowden, who was known to have used the service.

Possession of the SSL keys would in principle allow the FBI to intercept any and all traffic to and from LavaBit.com. It would also allow for the impersonation of LavaBit's servers, in a way which would be undetectable, allowing for material not originating from LavaBit.com to be sent to its users.

The ability for governments to forcibly acquire SSL keys entirely undermines the type of security that SSL is supposed to provide - i.e. ensuring that the information content of a communication is only available to the intended parties, and that those involved in the communication are who they say they are.

Cost to Business

The revelations about the extent and reach of the U.S. surveillance apparatus is expected to harm the profitability of businesses involved in cloud-computing which host their data within the United States' jurisdiction. A recent report by the Information Technology and Innovation foundation puts the cost (in terms of lost revenue) to the U.S. cloud computing industry at $22 to $35 billion over the three years from August 2013 [6].

Security breaches have a similarly chilling effect on business revenue and the broader economy. The PlayStation Network hack [45] alone is said to have cost Sony $170 million USD [52], while identity theft in general is said to cost the US economy over $56 billion USD per year (as of 2012) [8].

3.3 Summary of Current Literature

This chapter has summarised the relevant literature, and established the key techniques of data tainting and cryptography, including the storage and exchange of sensitive data. It has also established a number of key problems with, and gaps in those techniques, especially as they relate to their operation in a web-browser.

It has also established the current state of attempts to provide privacy-preserving services to internet users, some of their deficiencies, and the need to improve upon them. On this basis, the following chapter will define an abstract solution while will be expanded throughout the remainder of this thesis, in the specific context of JavaScript enabled web-browsers.

Chapter 4

Solution Overview

Here we present an overview of the model solution to the problem of how web services can preserve the privacy of their users. It is intended to inform the discussion in Chapters 5, 6, 7, and 8.

Our base assumption is that an individual or organisation wishes to create a web service which - insofar as possible - preserves the privacy of its users, and moreover offers some guarantees both to the service creator, and to the users of the service, that this privacy is preserved.

If the service creator does not wish to preserve their user's privacy, then there is no point in their utilising the solution presented in this paper. A third possibility exists, whereby the service creator wishes to *appear* as though they are preserving privacy in an attempt to dupe their users into a false sense of security. I would note that this could barely produce a worse privacy outcome than the one which already exists for web-service users at the moment; nevertheless our solution treats this case as a security breach, and deals with it appropriately.

Chapter 5 provides a model of how one might set up a service which accomplishes this goal. Under the correct operation of this model all sensitive user data is encrypted on the client before it is sent to the service (if it is sent at all). The decryption mechanisms are never conveyed in any usable form to the service, resulting in perhaps the most important property of our model - that the service cannot read the sensitive data, and therefore neither can anyone else on the service side of the equation, nor anyone in between.

Executed correctly, this property results in peace of mind for both the users and the service provider. From the user's perspective, they are assured that the service provider cannot disclose their sensitive information to any third party, willingly or otherwise. From the service provider's perspective, they are assured that even if they are compelled to turn over user data, or suffer a security breach in which their data stores are compromised, such data is all but meaningless in the form that they posses.

However, additional measures are required if the service is to retain broad usability comparable to most popular web services, while still offering these guarantees. Most services which currently offer these sorts of features require the user to explicitly download, install and/or configure specialist software in order facilitate the security mechanisms. That is an outcome we would expressly like to avoid, as many users are either not willing or not able to take those additional steps to protect their privacy.

If we are to avoid forcing users to install separate software in order to protect their privacy, then we must find another way to deliver these protections. Considering we are discussing web services, the most obvious delivery mechanism is via the web browser, which in principle

already possesses all the capabilities we need - that is, a user-friendly method of content delivery and a programming language (i.e. JavaScript) which is powerful enough to implement encryption in. However, this raises its own problems.

Firstly, and most importantly, web-pages and all of their associated content are supplied directly from the service (or wherever else they deem to load them from) when the user requests them, and (caching aside) this content does not tend to persist between requests. This means that the service is supplying to the user the cryptographic code (in the form of JavaScript) that is intended to protect the user's privacy, and that the service may modify this code at any time, without explicitly notifying the user. Obviously this is a serious security problem, given that a compromised service is the primary threat we wish to defend against.

The issue is not insurmountable, however. In order to operate under such a model while still guaranteeing the privacy of users, we need to verify three things:

- That the code the client receives is the code that the service intended to send; that is, it has not been tampered with by some unauthorised third party.
- The portion of the code that the user receives which implements the cryptographic and privacy preserving functions is the what the user expects to receive. That is, the service provider has not modified the fundamentals of the code which protects the user's privacy.
  - It is assumed that there is some reasonably standardised implementation of the cryptographic and privacy preserving functions, which can be checked against.
- The portions of the code which implement the actual web service - as distinct from the code which implements the cryptography and other privacy preserving functions - do not violate the user's privacy, nor do they do anything which could subvert the measures put in place by the cryptographic and privacy preserving functions.
  - It is assumed that the code which implements the cryptography and other privacy preserving functions does not in itself have the capability to violate the user's privacy.

It is important to note that all of the above are *checks*. If any of them fails, then is is possible that the user's privacy is being violated. However, it is possible for all of them to succeed without actually checking them[1], thus still in practice ensuring privacy.

The service cannot be relied upon to supply a way to check these things, for the same reasons we cannot completely rely on it to supply code which is guaranteed to preserve the user's privacy. Because we wish to maintain the service's usability, we also cannot rely on all users installing something which performs the checks.

However, what can (statistically speaking) be relied upon is that *some* users will install software if it guarantees their privacy. If we assume that the service cannot distinguish users who have installed such software from those who have not[2], then the service provider (or anyone else) would be unable to change their code in a way which violates privacy en mass, without someone noticing and making that information public[3]. In the case of trying to ensure privacy against server/service compromise, knowing that the violation exists is extremely useful, since if the data is encrypted then ceasing to use the service unless/until the issue is fixed is an entirely valid way for a user to protect their privacy and data.

[1] This will become important in a moment.
[2] Without some form of intervention this assumption is not actually true, but we will deal with that problem later.
[3] This approach belonging to the Don Chipp model of security; namely "Keep the bastards honest".

How to deliver software to perform such checks is a fairly dull question. Browser plugins cannot be tampered with by a service; they offer the integration with the browser necessary to perform the checks above; as we are not performing any sort of cryptography (particularly key-generation) within the software, we do not need the access to the operating system that an independent program provides; they also tend to be easier to install than an independent program. For these reasons, delivering these checks via a plugin seems the best route. If we wished to check code running in some environment other than the browser, then an independent program would likely be necessary. The only constraint we must place on such software is that its source-code be available, so that its correct operation can be independently checked and verified.

Chapter 7 will describe the details of software capable of performing the necessary checks.

Chapter 5

Service Model

In this chapter, we present a generalised model of how one could set up a web service with the intent to protect the sensitive information of the service's users. We will particularly focus on the case where the client-side portion of the service consists of a web page (or pages), including JavaScript code and any resources required by the page (images, stylesheets, etc), as this is by a large margin the most prevalent current method for delivering web services. However, the model (with a few provisos) could potentially be extended to other similar delivery methods, such as "apps" on mobile phones and tablets, and possibly further afield. Some of the model has been adapted from work done in industry - particularly that of Mega [24] and CryptoCat [20].

The model is intended to be verifiable by the client; that is to say that it is intended for it to be possible for the client to check that

- The client-side portion of the service is not disclosing any potentially sensitive information to any third party (including the server-side portion of the service).
  - The corollary of this is that all information disclosed to the service or to third parties is either not sensitive, or is encrypted.
- The client-side portion of the service has not been tampered with by a third party; that is to say, what is sent to the client is what the service intended to sent to the client, and what the client expected to be sent.
- All sensitive information which one user decides to make available to another user via the service is made available only to that user, and not to another user, to the server, or to a third party.
- The client-side portion of the service does not contain code with the potential to subvert any of the privacy preserving measures.

The actual process of how we can handle the verification is detailed in Chapter 7.

Potential adversaries of, and deviations from the model are dealt with in Chapter 7, as opposed to in the model itself.

5.1 Software/Website Model

We will model our service as a typical multi-page HTML+JavaScript-based website, accessed via a web-browser.

5.2 Actors

There are three active participating entities in our model.

- The user.
- The client.
- The service.

We consider the intermediaries between the client and the service (i.e. ISPs and internet traffic carriers) to be passive entities, merely responsible for passing data from one to the other. As both the connection and any sensitive data travelling on it is encrypted, there is very little they could actively do anyway.

Similarly the server - and any supporting software running on it, such as web servers and database servers - are assumed to perform their intended functions correctly when interacting with the client and the service.

5.3 Connection

- All communications between the client and server are assumed to take place over a secured connection (e.g. HTTPS).
- Any intermediaries are assumed not to be able to decrypt data sent between the client and server due to the use of HTTPS. There is the possibility that they will collect meta-data about the transmission, but attempting to avoid that is a) likely impossible, and b) outside the scope of this project.

5.4 Encryption Scheme

Our security model uses a moderately complex encryption scheme, visualised in Figure 5.1.

All encryption described under this scheme is performed on the client (e.g. in the web-browser).

The entities involved in this encryption scheme are:

- The user.
- The user's password.
- A symmetric master key.
- An asymmetric public/private key-pair.
- Many symmetric local keys.
- Many instances of distinct sensitive data.

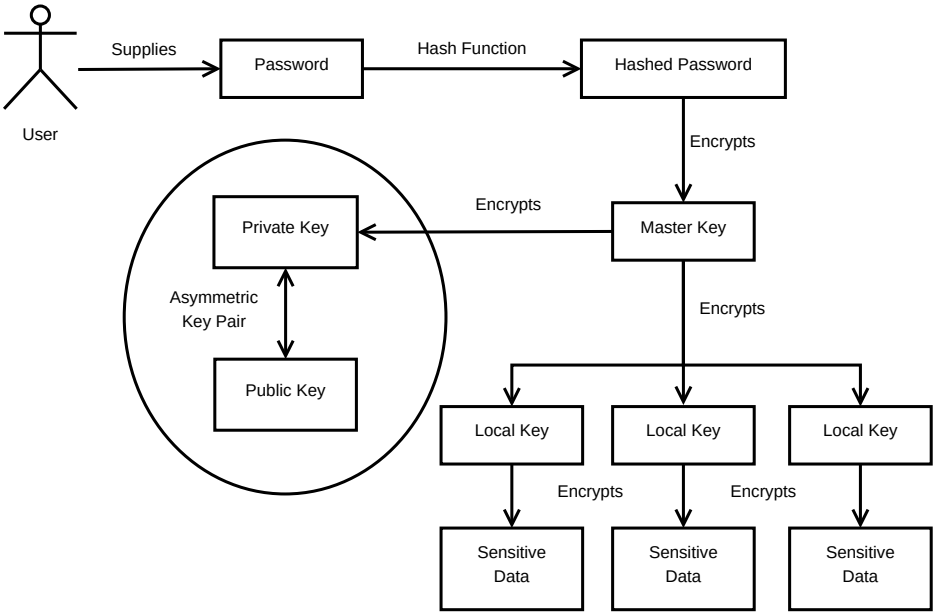

Figure 5.1: Diagram of encryption scheme dependencies.

Under this model, the user provides their password to the encryption software (e.g. via a password input in the browser). The password is then hashed, and this resulting hash may then be used to encrypt/decrypt the master key.

The master key is a symmetric key, which is used to encrypt/decrypt the private key, and all local keys.

The local keys are symmetric keys, which correspond one-to-one with an instance of sensitive data, which they may be used to encrypt/decrypt. An instance of sensitive data may be any logically distinct piece of information, such as the body of an email, a PDF document, or a post on a social network.

The public/private key-pair may be used to encrypt/decrypt information that the other key in the pair decrypts/encrypts, as is typical under asymmetric encryption schemes. The public key is never encrypted, since it is intended to be publicly available.

The public/private keys are necessary in order to send sensitive information to other users. The local keys are necessary in order to only make available specific data. If all sensitive information was encrypted using only one key, it is conceivable that another user could decrypt all of that data, rather that just the instance that was intended to be made available.

The master key could potentially be replaced by the public/private pair. However, asymmetric encryption schemes tend to be computationally much slower than symmetric ones, so it is preferable to minimise the use of the asymmetric keys to only when they are actually needed.

5.5 Common Elements

All pages and procedures are assumed to have the following steps in common:

1. The client uses a JavaScript-enabled web-browser or similar program to make an HTTP request to the server for the appropriate page.
2. The server responds, sending the corresponding HTML.
3. The client interprets the HTML, and makes additional requests to the server as appropriate for required resources (images, CSS, JavaScript etc).
4. The server responds by sending the appropriate resources.
    (a) Among the JavaScript sent to the client is code for performing the necessary cryptographic operations.

Figure 5.2 provides a visual representation of these steps.

Users may optionally install a plugin in order to verify that their privacy is ensured. This is represented appropriately throughout the diagrams for this model.

5.6 User Signup and Key Generation

The following describes the essential steps under the model for performing user signup and encryption key generation . Figure 5.3 provides a visual representation of this process.

1. The client requests and receives a sign-up page.
2. On the client, a public-private key-pair is generated (e.g. by use of the RSA algorithm [38]).
3. On the client, a randomly generated symmetric "master key" is created.
4. On the client, the private key is encrypted using this master key.
5. On the client, the user fills out a web-form, supplying at minimum a unique identifier (e.g. a username, or email address) and password.
    (a) This may happen concurrent to steps 2-4.
6. The master key is encrypted using a hash derived from the password.
7. The public key, the encrypted private key, and the encrypted master key and are all transmitted to the server, along with any other signup information. These are then stored on the server.
    (a) The unencrypted private key, the unencrypted master key, and the password (or anything derived from it) are *not* transmitted to the server.

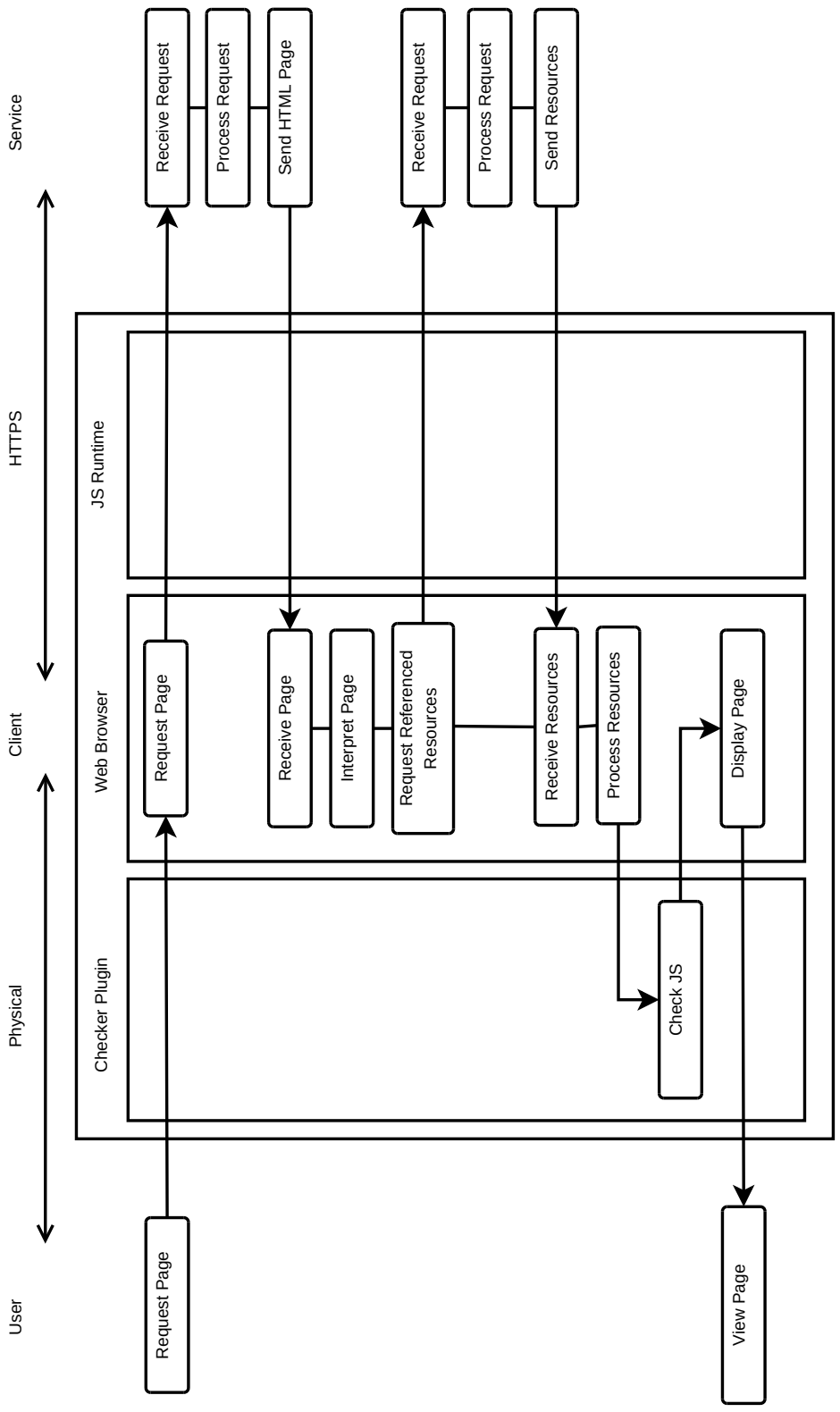

Figure 5.2: Common Elements

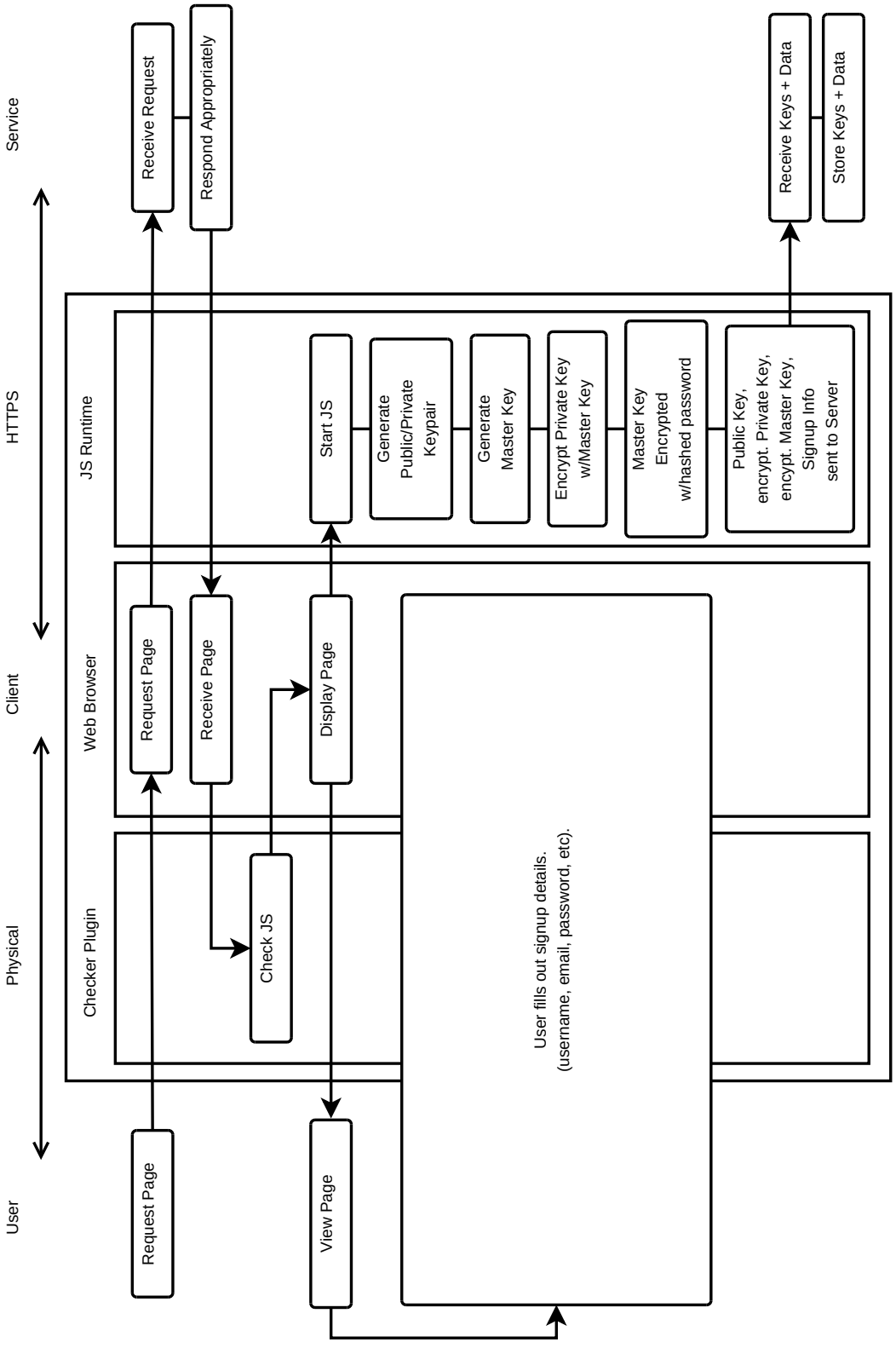

Figure 5.3: User Signup and Key Generation

5.7 Authentication and Log-In

The following describes the essential steps under the model for authenticating the identity of a user, and allowing them to log in to the service. Figure 5.4 provides a visual representation of this process.

1. The client requests and receives a page containing a login form.
2. The user fills out the log-in form with their unique identifier and password.
3. The user submits the form, which causes the client to send the unique identifier to the server.
    (a) But *not* the password.
4. The server responds with the encrypted private key, encrypted master key and public key corresponding to the unique identifier, and a random session token which has been encrypted using the public key of the corresponding unique identifier.
5. The client decrypts the master key using a hash of the supplied password, then decrypts private key using the master key, and finally decrypts the session key using the private key.
6. The decrypted session key is then sent to the server, and the user is considered authenticated if the key matches the original on the server.
7. The session key is then set as a cookie, so that the session may be tracked across pages.
8. The decrypted private key, master key and public key are saved to the JavaScript `localStorage` object.
    (a) This is so that they persist across pages, but in a manner not opaquely available to the server (in contrast with cookies).

5.8 Storing Sensitive Information

The following describes the essential steps under the model for enabling a user to store sensitive information in an encrypted form with the service, while ensuring that such information is not readable by the service. Figure 5.5 provides a visual representation of this process.

1. The client requests and receives a page which may be used to supply and encrypt sensitive information.
2. The user supplies the sensitive information they wish encrypted and stored to the JavaScript runtime, in some manner applicable to the type on information.
    (a) Different types of information may use different forms of input. The specifics are not relevant, other than that it reaches the JavaScript runtime in a usable fashion.
3. On the client, randomly generated symmetric keys (the “local keys”) are created for each distinct piece of information to be encrypted.

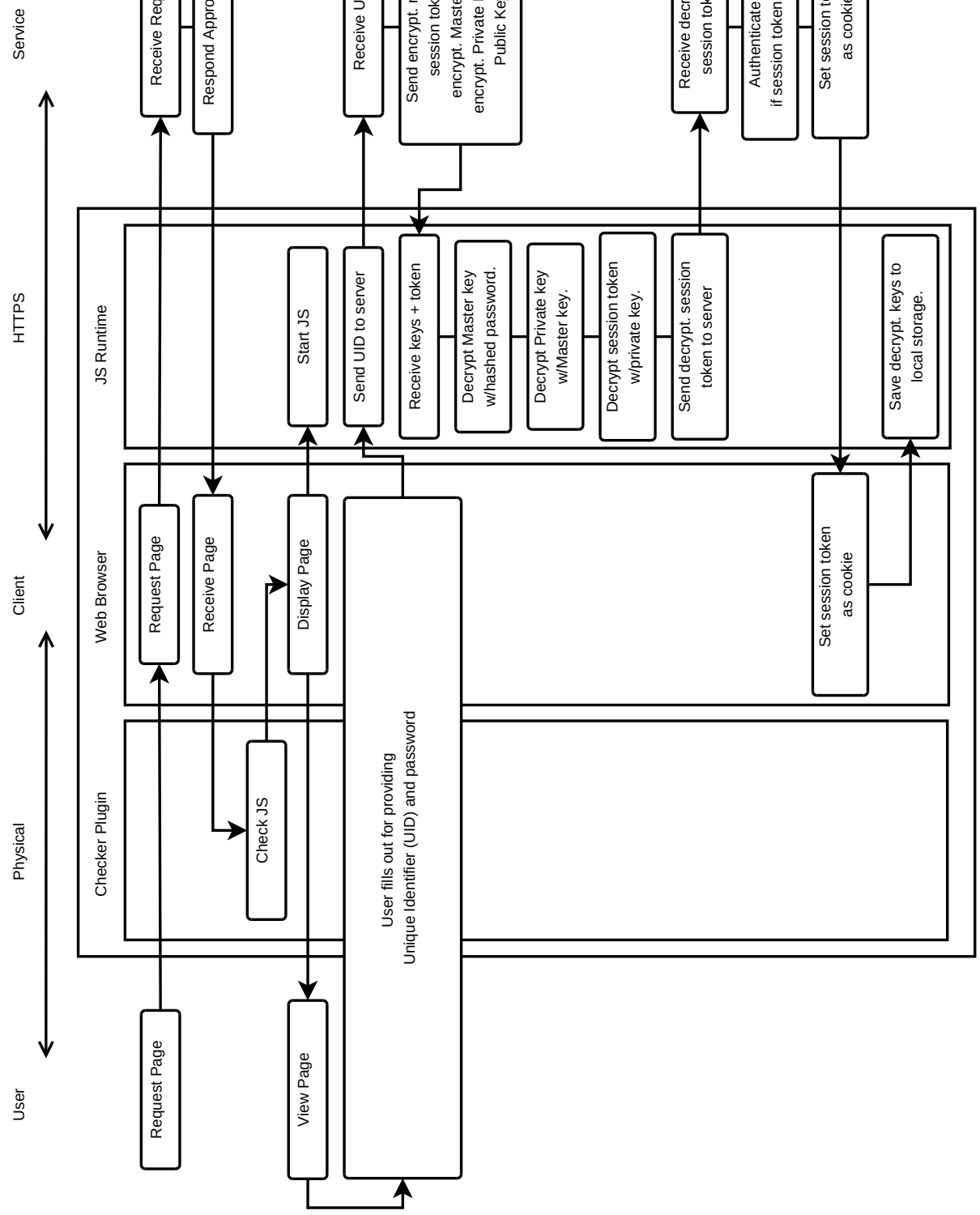

Figure 5.4: Authentication and Log-In

4. The JavaScript runtime encrypts the information using the local keys.
5. The encrypted information is transmitted to the server.
   (a) The unencrypted information is *not* transmitted to the server.
6. The local key(s) are encrypted with the master key, and transmitted to the server.
   (a) The unencrypted local keys are *not* transmitted to the server.

5.9 Retrieving Sensitive Information

The following describes the essential steps under the model for enabling a user to retrieve sensitive information that was previously encrypted and stored with the service, either by that user, or by another user that has made the information available to the first user. Figure 5.6 provides a visual representation of this process.

1. The client requests and receives a page which may contain encrypted sensitive information.
2. This may contain encrypted sensitive information, including the encrypted local key(s) corresponding to said information.
3. In place of, or in addition to the encrypted information sent in step 2, the client requests and receives encrypted sensitive information, including the encrypted local key(s) corresponding to said information.
4. The encrypted local keys are decrypted using the master key. The encrypted information is then decrypted using the local keys.
5. The information is made available to the user in whatever format is appropriate to the information.

5.10 Making Sensitive Information Available to Other Users

The following describes the essential steps under the model for enabling one user to share sensitive information with one or more other users.

1. The client requests and receives a page which facilitates making sensitive information available to other users.
2. The user indicates the specific information they would like to make available to other users.
3. The user indicates the specific user(s) they would like to make the information available to (using the unique identifier associated with each user).
4. The public key corresponding to each of the specified unique identifiers is requested from, and supplied by the server[1].

[1] There is obvious potential for a man-in-the-middle attack here, which is addressed in Section 7.10.

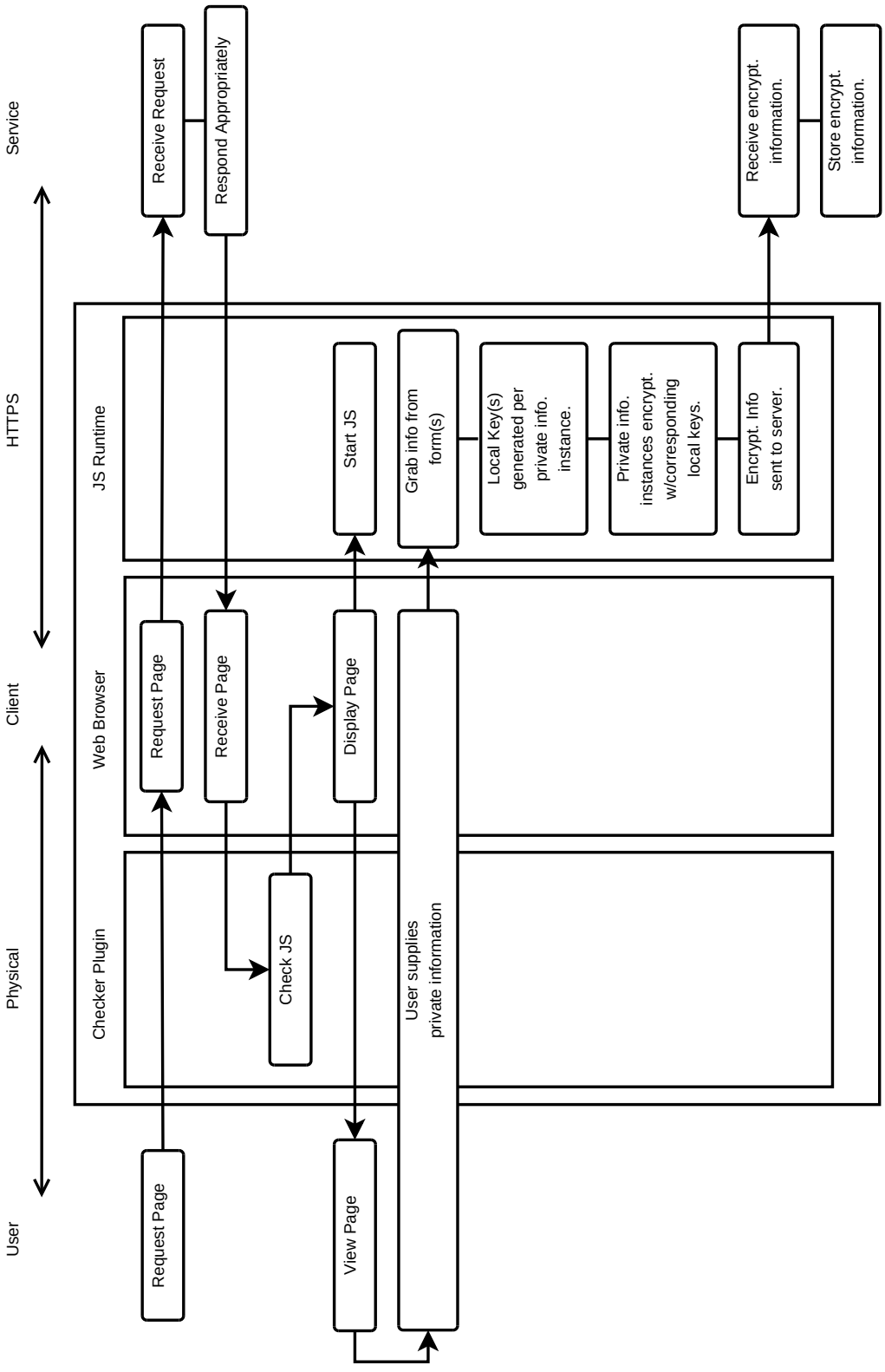

Figure 5.5: Storing sensitive Information

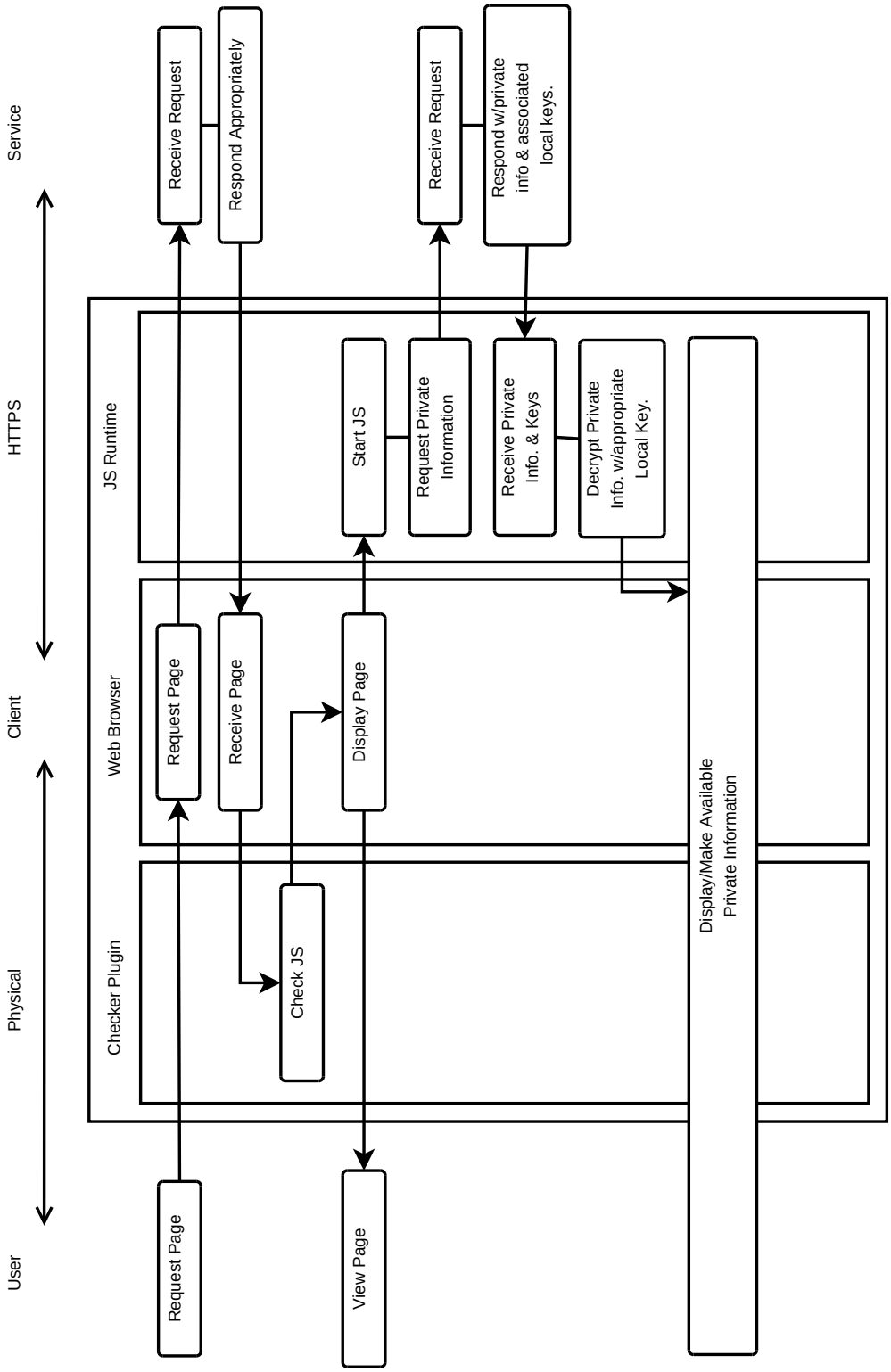

Figure 5.6: Retrieving sensitive Information

5. The local keys corresponding to each of the specified pieces of information are retrieved by the server, and decrypted using the user's master key.
6. The local keys are then re-encrypted using the retrieved public keys.
7. These re-encrypted version are then sent to the server.
8. On the receiving end, the process works as described under the "Retrieving Sensitive Information" section.

5.11 Specification of Sensitive Data

One important aspect that must be noted is this: what specifically is considered "sensitive data" as utilised under this model is defined *by the service*. This means that a service is only as privacy-preserving as its creator wishes it to be. Once data *is* designated as being sensitive (and therefore encrypted), it cannot then be undesignated as such, and at that point such data will never be available to the service.

This is necessary so that the the service can delineate between sensitive data and service-critical data, but it must be reiterated, these designations are defined by the service, not the user.

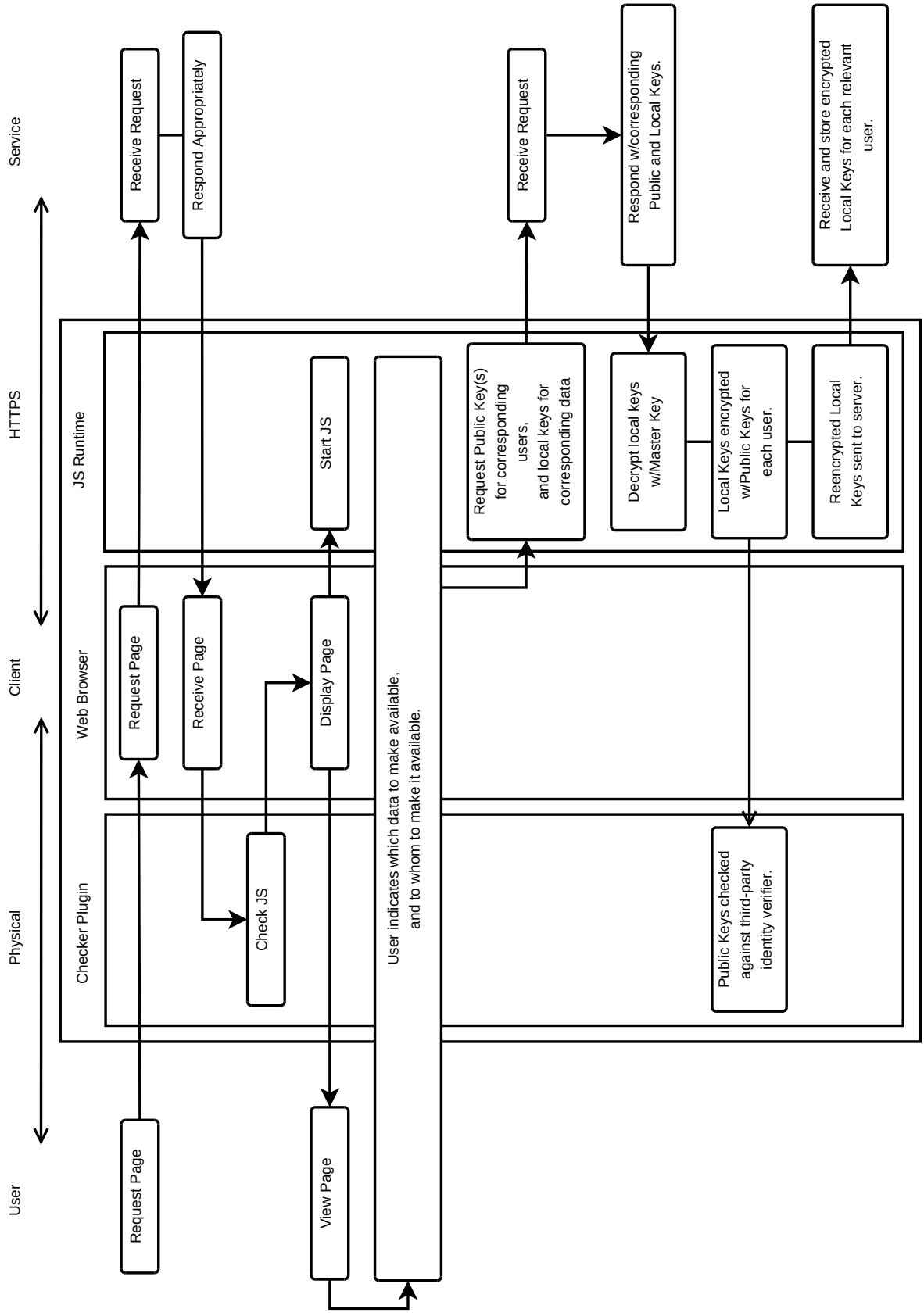

Figure 5.7: Making sensitive Information Available to Others

Chapter 6

Implementation Details

In this chapter, we will describe portions of an implementation of this model which are necessary for the plugin to function correctly. For sections involving JavaScript, jQuery's [18] notation will be used for conciseness. This is not strictly necessary for implementation.

6.1 Cryptography

We require an appropriate implementation of both a symmetric key algorithm (e.g. AES [47]) and an asymmetric key algorithm (e.g. RSA [38]) in JavaScript. These implementations must provide encryption, decryption, and key generation functions for both algorithms.

The psuedo-random number generator is assumed to be supplied by the browser, in the form of the `window.crypto.getRandomValues` function [32], or failing that any other appropriate cryptographically secure random number generator.

6.2 Sensitive Information Designation

In the case of text, we require that all sensitive information included as a part of the initial HTML page be designated like so:

```
<div class="sensitive" key="encrypted-local-key-here">
    Encrypted information here.
</div>
```

The "`sensitive`" class is a flag for the cryptography library to indicate both that the tag contains sensitive information, and that it should be decrypted for the user. As per normal HTML rules, the class attribute may contain other values in addition to "`sensitive`", and this will not interfere with the decryption.

The *key* attribute is a non-standard HTML attribute, used here to store the encrypted version of the local key corresponding to the sensitive information enclosed by the tag.

Finally, it should be noted that there is no requirement that the tag used be a `div` tag - that is simply used here as an example. We do require that there be a tag wrapping the sensitive information though, and that such a tag be the inner-most tag, relative to the encrypted information.

Non-textual information should be similarly designated, but in a format appropriate to the type of information.

6.3 Decryption

There are two primary methods by which sensitive data may be decrypted, beyond the assumed basic cryptographic `decrypt` functions.

The first method applies if an encrypted form of the data is present in the HTML page at load time, as described above. In that instance, we use the following function:

```
//On document ready
$(document).ready(function() {
    //For-each tag marked "sensitive".
    $(".sensitive").each(function() {
        //Do not want to decrypt input.
        //(See below)
        if($(this).not("input") {
            //Grab the cyphertext
            var cypherText = $(this).html();

            //Grab the key
            var localKey = $(this).attr("key");

            //Decrypt the local key
            //(masterKey is assumed to be in scope/retrievable)
            localKey = decrypt(localKey, masterKey);

            //Decrypt to cleartext
            var clearText = decrypt(html, key);

            //Replace cyphertext with cleartext
            $(this).html(clearText);
        }
    }
});
```

The second method applies if an encrypted form of the data is loaded in after the initial page load (e.g. via an AJAX call), and needs to be added to the DOM. In that instance, we use the following function:

```
//DOMReference contains a reference to an
//existing DOM node.
function (cyphertext, key, DOMReference) {
    //Decrypt the local key
    //(masterKey is assumed to be in scope/retrievable)
    localKey = decrypt(key, masterKey);

    //Decrypt to clearText
    var clearText = decrypt(html, localKey);
```

```
    //Ensure that the wrapping DOM node
    //is marked as sensitive.
    DOMReference.addClass("sensitive");

    //Ensure the wrapping DOM node
    //has the key attached.
    DOMReference.attr("key", key);

    //Replace DOMReference's inner HTML
    //with the cleartext.
    DOMReference.html(clearText);
}
```

6.4 Creating & Encrypting Sensitive Data

Input fields who's contents are intended to become sensitive information must be designated as such. For example:

```
<form id="aSensitiveForm" action="A_URI">
    <input type="text" name="foo" class="sensitive" />
    <input type="text" name="bar" class="sensitive" />
    <input type="submit" value="Submit" />
</form>
```

The implementation must be careful that the automatic decryption does not accidentally try to decrypt a form due to the "sensitive" class.

Upon submission of these input fields by the user, their contents should *NOT* be sent to the server. Instead it should be encrypted according to the scheme described in Sections 5.4 and 5.8, before being sent from within the JavaScript runtime. Example code for doing this is below. In practice, it would be more appropriate to serialise the form properly instead of writing back to it.

```
$("#aSensitiveForm").submit(function(e) {
    $(this).children("input.sensitive").each(function() {
        //Create local key for input field.
        var local = createLocalKey();

        //Encrypt field
        var cypher = encrypt($(this).val(), local);

        //Encrypt local key
        var eLocal = encrypt(local, masterKey);

        //Write back encrypted data
        //(For example purposes only. Should not
        //be done in practice).
        $(this).val("key:"+eLocal+",val:"+cypher);
    });
```

```
        //Submit via Ajax.
        $.post(
            $(this).attr("action"),
            $(this).serialize()
        );

        //Stops the browser from submitting the form
        e.preventDefault();
    });
```

Obviously, input elements which are not marked as needing encryption should not be encrypted.

Chapter 7

Plugin Model

In this chapter we will describe a browser plugin intended to allow users to perform the checks outlined in Chapter 4. We will describe all the relevant elements required to analyse JavaScript code, in order to determine whether it has the capability to "leak" sensitive information. We will also describe how to perform the relevant checks in order to ensure that the JavaScript code is actually analysable; that the integrity of the cryptographic code is maintained; that the application code has not been tampered with; that an exchange of data between users is as secure as it reasonably can be; as well as a number of other checks and constraints which are relevant in practice.

If any of the checks described in this chapter fails, or we encounter a case which is disallowed under this model, two things must happen. Firstly, the plugin must suspend the execution of JavaScript code on the page that has failed the checks. Secondly, the user must be informed of the failure, along with any relevant information about the failure. At least two options must then be available to the user: Allow the continueed execution of the code, or permanently suspend execution. Behaviour beyond that is left as an implementation decision.

7.1 Static Data Tainting for JavaScript

We may utilise the concept of data tainting [49] in order to detect cases where sensitive data may be accessed from within JavaScript. Briefly, data tainting is a form of control-flow analysis [1] whereby we mark a particular set of initial functions and operations as being tainted, and then any operations which depend on something tainted also become tainted. In our case, we mark functions which have the potential to return sensitive information as being tainted. These taint markings can then be used to detect whether sensitive information is being used inappropriately; for example, we could check if the arguments to an AJAX call are tainted, indicating that sensitive data could be transmitted to a server.

We will use the concept of *static* data tainting, as opposed to *dynamic* data tainting. The former involves performing a pre-runtime analysis of the code itself, while the latter involves analysing the code as it is running. Dynamic data tainting has the property of only analysing the code that is actually executed, whereas static analysis examines all possible execution paths. Which of those is preferable is a matter of point of view - the dynamic approach is more precise, in that it gives less false positives; however, the static approach allows the program to examine the *intent* of the program, even if that intent is not carried out. For our purposes, the choice to use static analysis is made largely because it is easier to describe and implement, and has a lower performance overhead.

In order to perform static taint analysis, we need four things:

1. The software's source-code.
2. A set of initially tainted functions.
3. A set of taint-sink functions.
4. A definition of taint propagation in JavaScript.

We have the first (in-browser JavaScript is inherently open-source). The third is relatively easy to define: AJAX calls, and anything else which could be used to communicate information to the server. The fourth point will be defined in Section 7.2, while the second will be defined in Section 7.3.

An important point to note is that the propagation model presented here attempts to minimise or eliminate false-negatives, which is to say that everything which *could* possibly depend on tainted data is marked as such. This unfortunately results in an abundance of false-positives. This is much less of a problem than in it generally is in taint analysis models, specifically because most such models attempt to protect the *program* from the *user*, while we are attempting to protect the *user* from the *program*. This places a great deal more flexibility in the hands of the developer to program around the constraints of the propagation model.

7.2 Taint Propagation Model

7.2.1 Tainted Entities

A *tainted entity* shall be defined as any JavaScript data construct (that is: variables, functions, objects, arrays, and anything else which may be represented by an identifier) which has been marked as tainted.

The set of tainted entities is tracked by the taint propagation model. All initially tainted functions (defined in Section 7.3) are marked as tainted, and considered tainted entities. Other entities are marked as being tainted or not tainted as the program is analysed using the propagation model.

7.2.2 Expressions

For our purposes, an expression will be defined as some chain of *operators* (+, *, -, /, &&, ||, etc) and *operands* (constants, variables, function calls, etc). If any of the operands is a tainted entity, then the expression as a whole shall be considered tainted, although operands which are not already tainted will not be marked as tainted as a result of this.

For example, if the variable `foo` is tainted, and `bar` is not, then the expression

```
42 * foo + bar;
```

will be tainted as a whole. However, the variable `bar` will remain untainted.

Function Calls

For our purposes, a function call is something of the form

```
func(arg1, arg2, ...);
```

where '`arg1, arg2, ...`' represents some a finite number of arguments to the function call, and `func` represents the function name. Each argument may be an expression in its own right. If any of the arguments are tainted, then the whole function *call* is considered tainted - although the function *identifier* is not marked at tainted as a result of this. The other arguments *may* become tainted (see Subsection 7.2.12). For example

```
func(taintedArg, untaintedArg);
```

if `taintedArg` was tainted, but `func`, and `untaintedArg` were not, then the call as a whole would be tainted, but `func` would not become tainted, and `untaintedArg` *may* become tainted, depending on the function.

7.2.3 Assignments

For our purposes, an assignment operation is anything of the form

```
identifier = expression;
```

If the `expression` portion of the operation is tainted, then *everything* in the assignment portion of the operation is also considered tainted. In particular, assume that in the following code listings, `x` is already tainted.

```
1 var foo = x;
2 foo = x;
3 foo[i] = x;
4 foo.bar = x;
5 foo = bar = x;
```

In line 1, the new variable `foo` is marked as tainted.

In line 2, the existing variable `foo` is marked as tainted, regardless of whether it previously was.

In lines 3 and 4 the variable `foo` is marked as tainted; `foo[i]` and `foo.bar` would technically also be marked as tainted, but this is subsumed by the marking on the parent entity.

In line 5, both `foo` and `bar` will be marked as tainted.

If an identifier is currently marked as tainted, but is assigned an untainted expression, then the tainted marking will be removed from that identifier from that point forward in the program (unless it is within a tainted scope, see 7.2.5), until it is assigned another tainted value.

Everything above also applies to the other assignment operators: `+=, -=, *=, /= %=, x <<= y, x >>= y, x >>>= y, x &= y, x ^= y, x |= y`.

7.2.4 References

Suppose we have a identifier which is really just a reference to another identifier. If one of these entities becomes tainted, the other must also become tainted. For example:

```
var a = {foo: "Hello"}
var b = a;
b.foo = taintedEntity;
```

In the above example, the variable `a` must be tainted if b is (and vice-versa).

7.2.5 Tainted Scopes

In general, if a block-level statement in some way depends upon a tainted entity or expression, then we create a *tainted scope* covering the entire scope of the statement, such that the results of all statements within that scope become tainted. Entities which become tainted within a tainted scope remain tainted after the scope ends.

The specific statements this applies to are:

- `if .. else`
- `for`
- `for .. in`
- `for each .. in`
- `while`
- `do .. while`
- `switch`
- `try .. catch .. finally`

Technically it could also apply to `with` statements; however, those are disallowed under this model (see subsection on With Statements).

7.2.6 If .. Else Statements

`If .. else` statements are of the form

```
if (conditionalExpression) {
    statement1A;
    statement2A;
    ...
} else {
    statement1B;
    statement2B;
    ...
}
```

with the usual proviso that the `else` section is optional.

If `conditionaExpression` is tainted, then we create a tainted scope across both branches of the statement, such that the results of all statements within both branches are considered tainted.

We must also consider the results if `conditionaExpression` is not tainted. If some entity is tainted in either branch, this that entity is considered tainted after the if-statement. However, the analysis of that entity in one branch cannot affect the analysis of it in the other branch. For example, suppose that the identifier `x` is already tainted, but `foo`, `y`, and `z` are not:

```
var foo = 0;
if(y > z) {
        foo = x;
        ...
} else {
        foo = 42;
        ...
}
```

Within the `if`-branch (i.e. when `y > z`) `foo` will be marked as tainted, and this taint may be propagated to other statements in the `if`-branch. However, `foo` is *not* tainted in the `else`-branch (i.e. when `y <= z`), and thus it may not taint other statements in the `else`-branch (assuming some other statement does not taint it further down that branch).

Furthermore, after the `if .. else` statement as a whole is analysed, `foo` will be marked as tainted, as it was tainted in at least one branch.

7.2.7 Switch Statements

Tainting of `switch` statements behaves similarly to `if .. then` statements.

```
switch (expression) {
        cases;
        ...
}
```

If `expression` is tainted, then we generate a tainted scope across the body of the `switch` statement.

If `expression` is not tainted, then each `case: ... break;` pair is treated in the same was as an individual branch of an `if .. else` statement. In the event of fall-through cases, the top-most case in the fall-through chain takes precedence in determining taint status.

7.2.8 Loop Statements

Loop statements, (`for`, `for .. in`, `for each .. in`, `while`, `do .. while`) behave quite similarly to one another, and to conditional statements.

While Loops

The simplest cases are the `while` and `do .. while` statements:

```
while (invariant) {
        statements;
        ...
}

do {
        statements;
        ...
} while (invariant);
```

In both cases, if `invariant` is tainted, then we generate a tainted scope across the body of the loop.

In **all** cases of loop statements, if the loop invariants are **not** tainted, it is still necessary to perform an analysis of the loop body more than once. This is due to the possibility that there exists an entity which is not tainted prior to the loop being analysed, but which becomes tainted during the loop analysis. Such an entity could conceivably taint entities which syntactically occur before it, during subsequent passes of the loop. For example:

```
var foo = untainted-entity;
var bar = untainted-entity;
while(untainted-invariant) {
        var baz = bar;
        bar = foo;
        foo = tainted-entity;
}
```

In this example, neither `bar` nor `baz` would not be tainted during an initial analysis of the loop, as neither the loop condition, nor `foo` is initially tainted. However, `foo` becomes tainted before the end of the loop, and thus `bar` would become tainted on a second loop pass loop pass. Similarly, `baz` would become tainted on a third loop pass. It is trivially possible to construct cases whereby some entity would only become tainted after an arbitrarily large number of passes, though such a number would be fixed for each specific loop instance.

Me must therefore repeatedly analyse the loop until we reach a point where the taint status of all entities within both the body of the loop and the loop invariant(s) do not change between analysis iterations (i.e. we reach the fixpoint).

Because of this, we also need to disallow *un*tainting of identifiers within loops. If this is not done, it is trivial to set up an infinite loop in the analysis. For example, the below is impossible to analyse without knowing how many iterations the loop will go for, as the taint status of `foo` and `bar` swaps with every iteration.

```
var foo = tainted-entity;
var bar = untainted-entity;
while(untainted-invariant) {
        var tmp = foo;
        foo = bar;
        bar = foo;
}
```

For Loops

In the case of a `for` loop:

```
for (initialisation; condition; final-expression) {
        statements;
        ...
}
```

If any of `initialisation`, `condition`, or `final-expression` are tainted, then we generate a tainted scope across the body of the loop.

In the case of `for .. in`, and `for each .. in` loops:

```
for (variable in object) {
        statements;
        ...
}

for each (variable in object) {
        statements;
        ...
}
```

If `object` is tainted, then we generate a tainted scope across the body of the loop.

Break & Continue Statements

`Break` and `Continue` statements within loop bodies are ignored, for the purposes of taint analysis. `Break` and `Continue` statements with an attached label (which functions similarly to a `Goto` statement in other languages) are not allowed under the model, as they make control flow analysis much too difficult.

7.2.9 Try .. Catch .. Finally Statements & Exceptions

`Try .. Catch .. Finally` statements are not handled under our model. This is due to the considerable complexity in analysing them. Specifically,

- Exceptions may be tainted.
- A `Try` block may contain an arbitrary number of sources of exceptions.
- Functions which throw exceptions may throw different ones based on different conditions, and not all of these are necessarily tainted.
- A `Catch` block has no syntactical way of distinguishing the types of exceptions it is receiving[1]. This is to say that all possible exceptions from a given `Try` block are caught in one `Catch` block.

[1] There does exist a so-called conditional catch block, but it is a non-standard feature, and not on a standards track. [31]

- Even if the `Catch` block could distinguish the type of an exception, there could be multiple sources of that type of exception.
- Data flow through the blocks cannot be determined statically, since exceptions mean control flow can jump out of the `Try` block mid-flow, and there are potentially multiple points at which this can happen.

This is not to say that it is impossible to analyse `Try .. Catch .. Finally` statements - there probably does exist some reasonable propagation model for them which doesn't allow for false-negatives; I merely have not found one. There is of course the trivial one where we just assume everything in the statement is tainted, but that is hardly useful or satisfying.

The closest I have come to a propagation model for these statements is:

- Assume all code in the `Try` block executes.
- If there are *any* sources of exceptions in the `Try` block, assume the `Catch` block executes.
- If there are any sources of *tainted* exceptions in the `Try` block, treat the `Catch` block as a tainted scope.
- Assume all code in the `Finally` block (if such a block exists) executes, after the other two blocks.

However, because of the complexities of the `Try .. Catch .. Finally` statements, I have been unable to properly check if this model is sufficient.

Exceptions, by contrast, are relatively simple - they can be handled as another type of `Return` statement, if they are not themselves wrapped in a `Try` block. However, a model for exceptions is not particularly useful without the corresponding `Try` statement to handle them.

A promising avenue of inquiry is a paper on "Analysis of Programs With Exception-Handling Constructs" [46] which attempts to deal with the analysis of exception handling constructs in Java. Java's exception handling is similar enough to JavaScript that their work can likely be carried over.

The problem is also trivially handled under a dynamic data tainting model.

7.2.10 With Statements

`With` statements are disallowed under our model. This is due to the fact that they create behaviour which is at best very difficult to analyse without actually executing it. Take for example:

```
var obj1 = {b: "Hello"};
var obj2 = {foo: "World"};
(function (a, b) {
    with (a) {
        b = "Goodbye";
    }
})(obj1,obj2);
alert(obj1.b); //alerts "Goodbye"
```

The above will alert "Goodbye". However, the function *cannot* be analysed in isolation, since in isolation it is impossible to know if the identifier `b` refers to the formal parameter of the function, or to a member of `a`.

Thankfully, `with` is a convenience statement[2], rather than one that is necessary to the use of the language, so we lose very little by disallowing it.

7.2.11 Events

Events are problematic because they are capable of running code outside the normal execution flow. They are also a source of potentially tainted data. Their use cannot be disallowed, as they are a vital part of creating interactive applications.

We therefore need to deal with three aspects of events:

- The attaching of events to tainted entities.
- Their interaction with the broader execution flow.
- Event bubbling.

In the first case, we say that the *entirety* of any event attached to a tainted entity is itself tainted, as is everything which occurs within the event handler.

If an event is attached to an untainted object, we must still consider its associated callback function. If the callback function does not interact with identifiers outside the scope of the function, then we can analyse it as normal. If, on the other hand, it does interact with out of scope identifiers, we must consider those identifiers to be tainted within the event handler if those identifiers become tainted at any point in the control flow analysis, even if they are later untainted. This is necessary, because we do not know at what point the event handlers will be triggered in the execution flow.

Finally, if an event occurs, and there is no handler attached to the inner-most DOM node that it occurs on, then the event will bubble up DOM nodes until it fines an appropriate handler, or until it reaches the outer-most DOM node without finding one [29]. The event object specifies the original target of the event. This original target may contain sensitive data. We must therefore consider the original target tainted.

7.2.12 Functions

JavaScript's functions are a heavily overloaded construct[3], so their behaviour with respect to data tainting deserves special care and consideration.

Function Calls

Firstly, as we defined in the section on Function Calls, if a function is *called* with tainted arguments, then its return value will also be considered tainted. This is on the principle that the result of any function a programmer would actually write almost certainly depends upon its arguments, and if this is not the case, they are always capable of rewriting it.

[2] And a rather poor one, by my own option.
[3] To put it mildly.

Pass-By-Reference

Secondly, we must consider the possibility of an object being modified via pass-by-reference. JavaScript does not possess an explicit pass-by-reference mechanism (there is nothing analogous to a C pointer). However, it will implicitly pass objects and arrays by reference. More accurately, it will pass a copy of their location in the runtime's memory, but the effect is still what is generally considered pass-by-reference. Thus we must deal with functions such as

```
function foo(obj) {
        obj.bar = taintedEntity;
}
```

Here, `obj.bar` becomes tainted within the function. Any object the function is called on should therefore become tainted outside the function.

In general, if a parameter is treated as an object or array within a function, and as a result becomes tainted, then we must treated the parameter itself as tainted. The result is that arguments to such parameters in a call of that function must also become tainted. For example

```
var anObject = {bar : "Hello", baz: "World"};
foo(anOject);
```

Before the call to `foo`, `anObject` is not tainted. After the call it is, because `foo`'s first parameter is tainted.

It is worth noting that even if a parameter is treated an object or array, only altering its members has an effect outside the function. Reassigning the parameter itself will not. Thus:

```
var obj1 = {foo : "hello"};
(function (a) {
        a = {foo: "goodbye"};
})(obj1);
alert(obj1.foo); //Will alert "hello"
```

This is important, as it means we may detect whether a parameter is an object or array by the way it is treated within the function - i.e. by use of `a.b` or `a[b]` notation - and if it is not treated as such, then it does not matter if it is an object or array.

Just as an object may become tainted via pass-by-reference, we must also consider the possibility whereby arguments taint each other. For example:

```
var obj1 = {foo : taintedEntity};
var obj2 = {foo : untaintedEntity};
(function (a, b) {
        var tmp = a.foo;
        a.foo = b.foo;
        b.foo = tmp;
})(obj1,obj2);
```

Here, `obj2` will become tainted by `obj1`.

The solution is to create a relation between the parameters of a particular function. For parameters $p_1 \dots p_n$. we say that $\forall i, j \in \{1, \dots, n\} : p_i \overset{\text{taints}}{\rightarrow} p_j \iff (p_j$ is an object or array, and p_i being tainted at the beginning of the function results in p_j being tainted at the end of

the function). Because of the idempotent nature of the taint property, and the way in which it spreads, it is sufficient to test each of the function parameters in turn - that is, mark one parameter as tainted and observe its effects upon the other parameters, then un-mark it and repeat for the next parameter.

Once this set of relations is computed, whenever we encounter a call to a function, we may observe which arguments are initially tainted, look up the relations for those arguments, and taint the other function arguments appropriately.

Out-of-Scope Variables

JavaScript functions will happily use identifiers from outside their own scope, if no such identifier is found within their scope. However, because such variables may be changed at many different points in the program, it can be difficult to analyse their taint status.

The simplest way to deal with such variables is to treat them as implicit parameters to the function, and analyse them as such. However, we must make the adjustment that such parameters do *not* need to be reference parameters in order to have an effect outside the function.

Once analysed in this way, whenever we encounter a call to a function which makes use of identifiers outside its scope, we can use the taint status of such identifiers at the point we encounter the function call when analysing its effects.

Function Arguments

JavaScript allows functions to be passed as arguments to function calls. Functions which utilise this feature are more difficult to analyse than those that do not, as a function's parameters may be tainted or not, rather than just the function itself. In order to analyse such functions, we defer their analysis until a call to said function is encountered, in which case we can substitute in the actual function used for the parameter. This does mean that we must re-analyse such functions at each call point, though some caching may be possible.

It *might* be possible to pre-compute the effects in the same way we do for function parameters. However, JavaScript's lack of an explicit or strong type system, combined with the variadic nature of its functions, makes that prospect something best left to the realms of theory for now. Additionally, it is still possible to pre-compute the effects of all non-function parameters.

Recursive Function Calls

As is always the case with static analysis techniques, recursion makes everything more difficult. All rules which apply to non-recursive functions also apply to recursive functions; however, we must also add in several additional rules in order to make them behave properly. As with loops, we disallow the untainting of identifiers within recursive functions. We then repeatedly iterate through the function calls until the taint status of all identifiers within the function stabilises (i.e. reaches a fixpoint).

In the case of mutually recursive functions (or chains thereof), we iterate through the recursion chain until the taint status of all functions in the chain has stabilised.

7.2.13 Objects & Classes

JavaScript encodes its classes as functions[4]. This makes analysing them somewhat challenging.

As noted above, if a property of a specific instantiation of an object becomes tainted, the whole object must be tainted. We must taint the whole object as opposed to just the property, because information could potentially be gained by the existence or non-existence of specific object properties. For example:

```
var sens = getSensitiveInformation();
var myObj = {};

if(hasInterestingProperty(sens) {
    myObj.yes = true;
    myObj.stillYes = true;
} else {
    myObj.no = true;
}

if(Object.keys(myObj).length == 2){
    //Object has interesting property.
    //Send this info to server.
}
```

As for classes, if any of the member functions or properties is tainted, or if any of the classes in the prototype chain are tainted, then the class itself is considered tainted, and therefore any time it is instantiated, the instantiated object will be considered tainted.

A class does not automatically become tainted by another class using it as a prototype.

7.2.14 Web Workers

The Web Workers API [33] provides a method of introducing genuine concurrency into JavaScript applications. While this is an *incredibly* useful feature, concurrency makes static analysis very difficult. We therefore disallow the use of Web Workers under the model.

7.3 Initially Tainted Functions

The other thing we require for this approach is to define a set of initial tainted functions. Broadly, this is defined as any function with the capability to retrieve sensitive information from the DOM. To the extent that I have been able to research, the only functions with this capability reside within the DOM API, although there are a *lot* of APIs available to the browser [27], so it is possible that there exist other functions with the capability. In either case, we shall focus our attention on the DOM API, as it is contains by far the most widely used set of functions known to have the capability, and the principles we shall use can easily be extended to other functions and APIs.

[4]If I may take a moment to personally comment on this situation: What on Earth were the language designers smoking when they decided that *that* was a good idea?

The aim is to define the set of initially tainted functions as narrowly as possible, as it is preferable to be as unrestrictive as is safely possible. I do not claim that I have actually *achieved* this, however - it is likely that there is room to either remove functions, or find a different but equally safe formulation.

7.3.1 Specific Functions

The set of initially tainted functions - along with their reason for inclusion in the set - is detailed in Table 7.2.

In addition to these functions, all accesses of the section of the `localStorage` object used to store the private key and master key are tainted. This is expressly so that the elementary cryptographic functions cannot be reimplemented in application code in order to create untainted versions of them, and also so the keys cannot be communicated to the server. Access to other sections of the `localStorage` object are not tainted.

The `document` object is specifically designated as not tainted, as otherwise almost every useful inbuilt method and object in the DOM and JavaScript runtime would be tainted by extension. This does preclude adding new properties and functions to the `document` object.

7.3.2 Preventing Data Laundering via the DOM

It is relatively simple to write a tainted value to the DOM and then read it back into JavaScript, storing it in an untainted variable in the process. It is also *not* easy to detect this case using static analysis techniques, as doing so seems to require (at minimum) knowledge of the DOM structure, due to the many different ways it is possible to read in the same DOM node. If we are to use purely static analysis, then it becomes necessary to treat all read operations from the DOM as tainted - or at least, all read operations that occur after a known tainted write.

Because of this problem, we need to include the functions listed in Table 7.3 in our set of initially tainted functions.

This is an instance where dynamic data tainting has a substantial advantage over the static case, as using dynamic data tainting would allow us to mark DOM nodes themselves as tainted, thus neatly dealing with this issue.

7.4 Putting The Usability Back In

If we were to mark all the functions mentioned in the Initially Tainted Functions section as tainted, without offering some form of workaround, it would be virtually impossible to write any meaningfully useful application in JavaScript that did not (under that model) appear to violate a user's privacy. We therefore need to offer some way that developers may use as many of those functions as possible, but in a way that we can be sure does not violate privacy.

A simple yet powerful approach is to wrap each of the tainted functions in function which - when run - checks to see if the attempted operation is actually trying to retrieve sensitive data. In the case where sensitive data would not be retrieved, the function would behave in the same was as its unchecked counterpart. In the case where sensitive data would be retrieved,

DOM API Object	Function/Attribute	Specific Risk
document	anchors	Returns an array of all `<a>` elements in the DOM
	body	Returns the `<body>` node, which may contain sensitive information.
	documentElement	Returns the `<html>` node, which may contain sensitive information.
	forms	Returns an array of all `<form>` elements in the DOM.
	getElementsById	Returns the element with a specified ID
	getElementsByName	Returns all elements with a specified name.
	getElementsByTagName	Returns all elements of a specific tag.
	links	Returns an array of all links in the DOM.
element	childNodes	Returns an array of the child nodes of an element.
	firstChild	Returns the first child node of an element.
	getElementsByTagName	Returns the child node so of an element that are a specific tag.
	id	Returns an elements ID attribute.
	innerHTML	Returns the HTML within an element.
	isEqualNode	Tests if two nodes are equal. Could be used to compare a node against one containing sensitive data.
	lastChild	Returns the last child node of an element.
	nextSibling	Gets the next sibling element of a node.
	nodeValue	Returns the value of an element, which may include sensitive text.
	parentNode	Returns the parent node of an element.
	previousSibling	Gets the previous sibling element of a node.
	removeChild	Returns the element removed.
	replaceChild	Returns the element replaced.
	textContent	Returns the textual content of a node, including all child nodes.
event	target	Can be used to access DOM nodes other than the one the event is attached to. These nodes could contain sensitive information.
	originalTarget	
	explicitOriginalTarget	

Table 7.2: Initially Tainted Functions

DOM API Object	Function/Object	Specific Risk
document	images	Returns an array of all `<img>` elements in the DOM.
	title	Returns the `<title>` element.
element	attributes	Returns an array of the attributes of an element.
	className	Returns the class of an element.
	getAttribute	Returns the value of a particular attribute of an element.
	getAttributeNode	Returns a node for a particular attribute of an element.
	hasAttribute	Checks if an element has a specific attribute.
	hasAttributes	Checks if an element has any attributes.
	hasChildNodes	Checks if an element has any child nodes.
	lang	Can be used to set and get the language code of an element.
	removeAttributeNode	Returns the attribute node that it removes. Same risks as getAttributeNode.
	style	Sets or returns the style attribute of an element.
	tabIndex	Sets or returns the tab index of an element.
	title	Sets or returns the title attribute of an element.
attribute	value	Returns or sets the value of an attribute.
	getNamedItem	Returns the attribute node with a specific name.
	removeNamedItem	Returns the attribute node that it removes.
	setNamedItem	Returned the replaced attribute node.

Table 7.3: Additional tainted functions.

Algorithm 7.1 A privacy-safe version of `getElementById`

```
function safeGetElementById(id) {
  var elem = document.getElementById(id);
  var cls = elem.className;
  if(cls.indexOf("sensitive") == -1) {
    return elem;
  } else {
    return null;
  }
}
```

the wrapper function would be required to not return the sensitive data. However, there are a number of different ways we could handle the specific behaviour of such functions:

The first is to fail silently - i.e. return `null`, or some equally non-informative value. This is probably not a good approach, as some of the tainted functions return `null` on failure anyway, so it would be difficult for a developer to distinguish why a function had failed. On the other hand, it would present a uniform mode of failure.

The second approach is to fail with feedback - i.e. return some special predefined value (or an exception) indicating failure, and type of failure.

In the case where an array or object is returned, only some elements of which may be sensitive, there also exists the option to remove the sensitive items, while leaving the rest intact. Again, we may either do so silently, or provide notification that items were removed.

Additionally, these possibilities are not mutually exclusive - it would be relatively simple to allow developers to specify the desired behaviour of the wrapper functions as one of the inputs to said functions.

Algorithm 7.1 provides a simplistic example of how we might go about wrapping the `getElementByID` function.

The result of this is the creation of a set of functions with similar functionality to our set of tainted functions, but providing the guarantee that no sensitive data will be returned by them. This allows our taint analysis model to work under the guarantee that the results of such functions are, and will remain untainted. This in turn provides developers with a way to access the DOM, without either the developer or the user being concerned about violation of privacy.

It is still necessary to allow use of the tainted functions, as there are legitimate cases where a developer might wish to retrieve sensitive information from the DOM. For example, one might wish to develop an index of sensitive data on the client side, such that it may be quickly searched later. Such an index could be encrypted and stored in the same way as other sensitive data, and updated as needed.

7.4.1 Dealing With Writes

As mentioned in the section on Preventing Data Laundering via the DOM, there exists a problem whereby tainted data may be written back to the DOM, and then read back in, in order to remove the tainted designation. This is not as easily dealt with as merely reading sensitive

data from the DOM is, as such data will not necessarily be designated as sensitive when it is written.

We therefore create another set of wrapper functions, this time to guard writes to the DOM. Specifically, such functions designate any data written via them as being sensitive. This enables our read functions to detect sensitive data written in this manner, and act accordingly.

On the data taint detection side, we outright disallow the use of unwrapped write functions on tainted data. The unwrapped functions can and indeed should still be used on untainted data.

This should not present a problem for developers, as it is always possible to simply keep the tainted data within the JavaScript runtime, instead of using a write-and-read-back pattern, and the sensitive designation - by design - should not have an effect on the way the DOM itself behaves.

7.5 Input Fields

The section on dealing with writes should ensure that any sensitive data written to an input field results in the field being marked as sensitive. However, we must additionally ensure that the contents all input fields which are marked as sensitive are encrypted before transmission. If this is not the case, sensitive data could be written to a form, and the form then submitted, resulting in unencrypted sensitive data being sent to the server.

7.6 Detecting Incidental Communication

AJAX calls are the primary method used to communicate with the outside world from within JavaScript. There may be other such methods of communication available to JavaScript - there are after all a *lot* of Web APIs [27], and I have not been able to check all of them. However, the analysis conducted with respect to AJAX can be applied to all such possibilities.

A more difficult problem is presented by the ability of the JavaScript runtime to communicate with a server via the loading of external resources. Consider the code in Algorithm 7.2, which loads a CSS file based on some arbitrary property of a piece of sensitive data. No sensitive data is actually communicated to the server, but the server is nevertheless able to discern the presence of sensitive data with that particular property on the page. Given that the service controls both the JavaScript code, and the structure of the page, there is virtually no limit to what information could be gained using this method.

Any external resources which are loaded during page load (i.e. before the JavaScript runtime starts) are safe, as their fetching cannot be used to communicate any sensitive information. We therefore only need to deal with the case of external resources being fetched after page load.

The most obvious way to deal with this issue is to interpose on individual HTTP GET requests, and detect whether they are loading something which is not already loaded. This would have the unfortunate consequence of needing to load in all external resources at page-load, regardless of actual need for them.

The alternative is to check all DOM nodes and HTML fragments before they are added to the DOM, to determine whether they use an external resource which has not already been

Algorithm 7.2 jQuery code to load an external CSS file.

```
$(document).ready(function() {
  // Lifts sensitive data from page
  var data = getSensitiveData();

  if(hasInterestingProperty(data)) {
    $('head').append(
      "<link type='text/css' rel='stylesheet'
      href='innocentCSSFile.css'>");
  }
});
```

loaded. This is quite a complex check; there are a lot of elements capable of loading in external resources - particularly since the introduction of HTML5 - and traversing and interpreting a DOM tree is itself a moderately complex exercise. However, it can be done. The upside to this approach is that such a check can be performed in JavaScript at runtime, meaning that we can provide checked and unchecked versions of functions which allow DOM modification, and apply taint analysis to the unchecked ones.

7.7 Ensuring Analysability

In order to analyse the JavaScript code for a particular webpage, we first need to ensure that the code is reasonably static - that is, the code that is analysed is the code that will actually be run on the user's machine. JavaScript possesses a number of mechanisms capable of subverting this assumption, which we must disallow.

Firstly there is the `eval` function, which accepts an arbitrary string, and will attempt to evaluate (i.e. execute) it in the JavaScript runtime environment. In other words, it enables arbitrary code execution. Even if the rest of the code is entirely static, if the server or some third party is able to manipulate the string(s) passed to a call to `eval`, then all other analysis is useless. Thankfully, it is rarely actually necessary to use the `eval` function (most of what it's used for can easily be done in some other way), and most coding standards and security policies recommend against its use. The same may be said for string arguments to `setTimeout` and `setInterval`, both of which allow for arbitrary code execution. For these reasons, we disallow the use of `eval`, and string arguments to `setTimeout` and `setInterval` under the model. Function arguments to `setTimeout` and `setInterval` are fine.

Secondly, it is possible to embed JavaScript code directly into the HTML page, through the use of `<script>...</script>` tags. While such tags are not strictly speaking problematic in their own right, they are largely unnecessary, as anything which can be done in them can be done in an external file, and it is easier to analyse the behaviour of a set of external files. The bigger problem with such 'inline' script tags, as opposed to ones which load in an external file, is that they can be added in after page-load with arbitrary contents. For example:

```
function foo() {
    alert("Hello");
}
```

```
var s = document.createElement('script');
s.innerHTML="function foo() {alert('Goodbye');}";
document.head.appendChild(s);
foo(); //Alerts "Goodbye"
```

For this reason, we disallow the use of inline script tags under the model.

For very similar reasons, the use of `onclick` attributes, and of links with an `href` attribute beginning with "`javascript:`" are disallowed under the model. Neither are necessary, in that everything they provide can equally be achieved in an external file.

Lastly, it is possible to dynamically load in external scripts by appending a new `<script>` tag to the DOM, with the `src` attribute pointing to some external URI, or by altering an existing tag. Again, there are few circumstances where this is actually necessary. Generally it is merely convenient. We therefore disallow loading in scripts not present on the initial page.

In short, only static external JavaScript files are allowed under the model.

7.8 Ensuring Indistinguishability

It is necessary to ensure that a service cannot distinguish between users who have the plugin described here installed, and those that do not. If we cannot ensure this, then a service could simply serve different code to the different user groups.

It is currently possible to access a list of browser plugins from within JavaScript via the `navigator.plugins` object [30]. Mozilla at least is planning to limit the ability of JavaScript code to enumerate through this list, instead restricting it to a direct query for specific plugins - however, that does not appear to have fully happened yet. In any case, we need to ensure two things:

1. That the JavaScript code is not specifically querying for the plugin described here.
2. That the JavaScript code is not enumerating through the list of plugins, at all.

It should be noted that the `navigator.plugins` object does not report things such as Firefox's AddOns (which are a distinct concept from plugins). However, it is also not a part of any standard (though it is widely supported), so its behaviour with respect to what it reports may vary across browsers.

7.9 Flash, Java, Silverlight, et al.

Flash has the capability to access the JavaScript runtime, and through it sensitive information contained in the DOM. Unlike JavaScript, the behaviour of Flash objects on a page is not easily analysed, because their source code is not readily readable. For this reason, Flash is incompatible with the model, and therefore its use disallowed [39].

Java applets face a similar issue, in that they are capable of accessing and manipulating the DOM [34]. Again, the behaviour of such applets is not easily analysed, and their use is therefore disallowed under the model.

I am not familiar enough with Microsoft's Silverlight to comment on its analysability, but it is certainly also capable of accessing the DOM. It - and any other third-party methods of delivering so called rich content - must be disallowed under the model if they are both capable of accessing the DOM, and not readily analysable. In the event that they *are* analysable, it may be possible to perform some analysis similar to what is described above with relation to JavaScript.

7.10 Data Exchange

As noted in Section 5.10, there is a very obvious potential man-in-the-middle attack related to the exchange of sensitive data between users. In particular, if the service is the entity facilitating the exchange of public keys between the users, then it can easily substitute the user's public keys with ones that it controls, allowing it to read the messages. Moreover, this attack is not easily detectable on the client side, because it can re-encrypt the messages with the keys the users expect.

The general approach to solving this problem is to perform the key exchange via a side-channel not controlled by the service [16, 3]. Unfortunately, to ensure usability for those not capable of doing this, the service does still need to perform the key exchange. However, we can *verify* key-identity pairings via an independent side-channel. Specifically, what we need is some third party which is trusted by the user that can attest to the authenticity of the public keys supplied by the service. This in turn requires that there exists a third party (or third parties) which possess those bindings.

The simplest way is for the service-provided JavaScript code to automatically report the username-to-public-key binding to a third party service - which we shall term the identify verifier - at signup. The JavaScript code can then use both the main service, and the identity verifier in order to verify the authenticity of the keys, and the plugin can check that this is actually occurring. This approach is still somewhat vulnerable, in that an adversary could potentially take control of both services. However, two services are safer than one, in that regard.

There are some additional measures that we can take within the plugin to ensure key authenticity. Firstly, individual users of the plugin are able to verify the authenticity of their own keys. That does not assist in the problem of data exchange, but it can be used to detect misreporting on the part of the service or the identity verifier, which could indicate a security/privacy breach.

Secondly, there is no technical reason to restrict ourselves to a single identity verifier. The more such services which exist, preferably in disparate geographical locations and legal jurisdictions, the less likely that any one adversary can compromise them all. Having multiple verifiers also allows the user to chose to use ones that they trust most.

Thirdly, there is no requirement that identity verification be a centralised operation. OpenPGP has demonstrated that so called webs-of-trust [16] are a viable method of performing key acquisition and verification. Such a system would actually be highly applicable in cases where users are only trying to communicate with a few individuals who are known to them - for example, in the case of a social network. There is also no reason that centralised and decentralised methods cannot work along side one another.

7.11 Ensuring Integrity of Application Code

In order to ensure that the application code received by the user is the code that the service intended to send, we require that the service do three things.

- Firstly, that it create a public/private key-pair, and attach the public key to its SSL certificate.
- Secondly, that it uses this public key to sign a hash of the top-level HTML page, and includes this signed hash in the response headers for the page.
- Thirdly, that all references to external resources on the HTML page include a "hash=*h*" parameter in their query string, where *h* is a hash of the external resource.

The plugin may then check the hashes of each of the external resources against the ones provided in the query string. This approach is taken directly from [35].

The first and second points ensure that the HTML sent is what the service intended to send. The third that all external resources are as intended.

7.12 Ensuring Integrity of Cryptographic Code

We assume that there exists a reference implementation of the cryptographic code in JavaScript, and all privacy preserving functions defined above. Said reference implementation must be available to both the creators of the service, and the creators of the plugin. Said reference implementation must be used verbatim by the service to provide its cryptographic and privacy preserving functions.

With these assumptions, the plugin may take a hash of the reference implementation file, and check for its presence at page load. There may, of course, be multiple different reference implementations for different purposes.

Additionally, we must ensure that the rest of the application code does not interfere with the cryptographic code. This is reasonable straightforward - we check that there are no assignments to any of the identifiers in the reference implementation.

7.12.1 Key Integrity

While reads from the master key and private key are tainted (see §7.3.1), writes to the master and private key by anything other than the cryptographic code are expressly disallowed. This is to ensure application code does not substitute these keys with ones it controls.

7.13 Detecting When to Use the Plugin

It is necessary to distinguish sites which intend to provide their users with privacy under the model described in this thesis, from those that do not.

The simplest way to do this is for the plugin to detect the presence of the reference implementation described above, and maintain a list of sites which it has previously detected its use on.

This list can be used to detect if a site suddenly ceases to use this model, which would suggest a potential security or privacy violation. Such a list could either be maintained locally, or on a central server.

7.14 Summary

In summary, the above attempts to ensure the following things:

- That we are aware of all cases of sensitive, decrypted user data entering the JavaScript runtime.
- That we are aware of all cases of sensitive, decrypted user data being written out to the DOM, and that these are marked appropriately.
- That sensitive, decrypted user data - and information derived from it - can be tracked throughout the JavaScript runtime, and that we can detect such data transmitted beyond the browser.
- That all sensitive user data is encrypted prior to being transmitted beyond the browser.
- That sensitive user data made available to other users is made available only to those users.
- That the JavaScript code sent to the browser is analysable.
- That the JavaScript code sent to the browser is what the service intended to send, and has not been tampered with.
- That the integrity of the cryptographic code, and the cryptographic keys is maintained.
- That neither the HTML code nor the JavaScript code sent to the browser attempt to subvert any of our security mechanisms.
- That those people who use plugin described here are indistingishable to an outside observer from those who do not.
- And that, in ensuring the above things, we do not make programming against this model unduly burdensome.

Chapter 8

Discussion & Further Work

8.1 Strengths and Weaknesses

8.1.1 General Discussion of Strengths

The greatest strength of the work presented in this paper is that it's an effective and usable way to ensure online privacy, not just for those who seek such privacy, but for all users. It is capable of providing strong protection against the problem of mass data theft by criminals, and the practice of mass surveillance, while *retaining* the ability of surveillance agencies and law enforcement to perform specific, targeted surveillance of individuals[1].

The model is also very generic in nature. I have focused quite specifically on the HTML+JS+Browser stack, as that is - by a wide margin - the most used method for delivering web services, particularly to those users who are not actively seeking to protect their privacy. However, the only thing really restricting the model to that set of technologies is the ability to access a webpage's JavaScript source-code. If that barrier can be overcome - for example, by publishing the source-code in other ways, or by analysing byte- or machine-code - then the model could easily be applied to other methods of web application delivery, such as mobile apps, and even (if it ever becomes necessary) general installed programs.

If a service-provider wishes to attempt to guarantee their user's privacy by legal means - for example, by operating in a country with strong privacy laws - then they can forgo everything described in the plugin model and rely on the legalistic protections instead. This is the approach taken by several companies in the field, including Mega [23] and Proton Mail [53], which have chosen to operate from New Zealand, and Switzerland respectively. Whether a technical or legal solution is preferable is really a matter of opinion.

The use of the service model alone is sufficient to guarantee user privacy in every case except those where an adversary gains control of the application servers or the service's encryption keys - that is to say, cases where an adversary can send arbitrary malicious code to the client.

There also exists another possibility which negates the need for many of the checks described in the plugin model. Briefly, the motivation behind that model is that we cannot *trust* the code coming from the server, and must therefore verify that it isn't doing something it isn't supposed to. However, under the service model, most of the operations performed on the server-end are store/fetch operations, as it is unable to do any complex computations over the encrypted data. This means that all the relevant application logic is present in the client

[1] That is to say, nothing in this thesis stops them from performing surveillance on specific individual users or client machines.

side code, with the result that a user - if they were so inclined - could simply drop in their own client-side application which they *do* trust, and which cannot be modified by the service, and thereby negate most of the remaining privacy and security concerns.

Such a possibility is not without its drawbacks - in particular, it removes the ability of a service-provider to easily deliver updates to its service to all users. However, that trade-off may be worth it, in the case of some services. Mega notably provides a public API for its service, and explicitly states on its website:

> "*If you don't trust us, you cannot run any code provided by us, which precludes opening MEGA in your browser and entering your login credentials. However, due to MEGA's end-to-end encryption paradigm, you can safely use client applications written by someone you trust.*" [22]

8.1.2 General Discussion of Weaknesses

The most glaring weakness of this thesis, and the ideas presented within, is the lack of a full implementation. While sections of it were implemented, such implementations were almost invariability prototypical and incomplete in nature.

Anyone seeking to use the work presented here would need to create a complete and robust implementation of both the service model and the plugin. The model described in Chapter 5 draws upon a wide body of existing literature, as well as some existing implementations [23, 20, 53, 35], and is presented with considerable confidence. The model presented in Chapter 7 - especially the parts relating to data tainting - have not been validated, and should not be relied upon as an authoritative implementation. It is presented firstly as a demonstration of the implementability of the declared requirements, and secondly as indication of how to go about a full implementation.

This is not to say that there is doubt about whether taint analysis *can* be performed in JavaScript; indeed, some ancient versions of Netscape actually had it as a browser feature [12], and there are a number of academic papers on the subject [12, 49, 51, 11]. However, all but one of the implementations and academic papers I have found on the issue utilise dynamic tainting. In hindsight, I suspect there is a good reason for that: while it certainly requires a little technically wizardry to set up - namely hooking into the JavaScript runtime - it has the benefit of it being *much* easier to be certain you are doing it correctly. The required technical wizardry is actually not too difficult, either; in Firefox at least, the Debugger API [28] provides hooks for plugins to both observe and manipulate a page's JavaScript runtime. The way it is used in both Firefox's internal development tools, and in Firebug, suggest that it possesses all the features necessary. Dynamic tainting would also neatly step around all of the parts of JavaScript that are not covered under the data tainting model presented in Chapter 7, such as `Try .. Catch` blocks, and `With` statements, although possibly not web workers. Were I to attempt this project from scratch, I would likely use dynamic tainting over static, or possibly a hybrid approach.

8.1.3 Strengths and Weaknesses of Taint Analysis Model

The model for taint analysis of JavaScript presented in Section 7.2 possesses definite room for improvement. In the first instance, there is a need for validation of the correctness and completeness of the model as a whole. That is to say, it needs to be thoroughly tested to ensure

all its components work, that all of its components work together, and that no corner cases or obscure sections of the ECMAScript[2] language specification have been missed. Whether such validation is done through rigorous systems testing, formal verification, or some other method really depends on how certain one needs to be about it.

Secondly, I believe there is considerable scope to define it more narrowly. The model given in Section 7.2 is defined in a way which is intended to ensure no false negatives (i.e. no possible violations of user privacy), and to have relatively simple rules. The result is that the model will report many false positives (instances where it thinks user privacy could be violated, but in actuality it is not). It is highly likely that different/more complex rules could be devised which still ensure a lack of false negatives, but which drastically reduce false positives. There is also the possibility of defining a model which permits some false negatives in exchange for greatly reduced false positives, without unduly risking user privacy - although such a model would inherently be more risky.

As for its strengths, I believe this is the first instance of the use of static taint analysis to detect privacy leakage in JavaScript. There exists some work on the individual components of that [11, 12, 17, 41, 42, 49, 51], but I am unaware of any instances of work on the topic as a whole. I also believe that while what I have described is certainly not perfect, the core concept of using taint analysis to detect privacy vulnerabilities is sound, and has the potential to allow a *lot* of web-service creators to guarantee their users' privacy.

Additionally, the specifics of the taint analysis model presented here are quite advanced and detailed, compared to much of the other work in the field. A notable point is the fact that much of the other work lacks the concept of a tainted scope, so for example if a tainted entity was used as the condition for an if statement, it would not taint the body of the statement - something which is vital for our privacy model to function correctly. I am also unaware of any other work regarding taint analysis which takes into consideration the interaction between JavaScript and the DOM, something which I have consistently tried to account for throughout (and which greatly complicates the model).

A benefit of the way the model is set up - especially with regards to the inclusion of untainted wrapper functions around tainted data sources - is that it is *much* easier to program around than it might seem on the surface. In particular, if a developer takes the time to isolate those sections of an application which *do* deal with sensitive data from those which don't, they can otherwise create the application "as normal".

It should be noted that while there is most definitely other work in the broader field (see the above references), I independently "invented" a lot of the material in Section 7.2 (and quite late in the project at that), and *then* went searching for other people's work on the subject. It was only by coincidence that I actually discovered the terms "taint analysis" and "flow-control analysis", which are the prevailing terms for the techniques in academia. This unfortunately meant that there was insufficient time to integrate much of the other existing work into this thesis.

8.1.4 Password Security

From a security standpoint, one of the more substantial issues with our model is the fact that the password is the weakest link in the encryption chain. It is vulnerable to all the usual attacks on passwords, including key-loggers, brute-force and dictionary-based attacks, and phishing.

[2]JavaScript is an implementation of ECMAScript.

In that respect, it is every bit as vulnerable as every other password-protected website. On the plus side, it is at least not *more* vulnerable.

8.1.5 Malicious External Resources and the Same Origin Policy

There exists the possibility that an external resource loaded into the HTML page could contain malicious JavaScript code, and that it could be loaded in such a way that this is not detected by the plugin. If it is loaded as a top-level page, then it could execute with full privileges, and potentially make off with user data or encryption keys.

Mylar presents a possible solution to this. Namely, the use of two separate domains to host the site's content. One hosts the top-level HTML page, and the other hosts all other resources. This would result in the browser's same-origin policy being invoked, meaning malicious resources would not be able to run with the privileges of the root page [35].

8.2 Implications for Practice

8.2.1 Password Recovery

Under the service model described in Chapter 5, the user's password is used as a "root" security token, in the sense that it is used encrypt the master key, which encrypts everything else. The password is also never stored with the server. An unfortunate side effect of this is that if the password is lost, the account's data is unrecoverable.

This risk can be mitigated by recommending to users that they back up either their master key or their password. However, that is very much a social patch to a technical problem, and not one that gels well with the "usable by everyone" goal of this thesis. Any possible solutions to this problem are therefore worth investigating.

8.2.2 Implications for Business Models

Given that a substantial chunk of the business models of internet companies depend upon having access to their user's data, we would be remiss not to consider the affect of our model of ensuring privacy upon their business models.

The unfortunate truth is that targeted advertising as it is practised by the likes of Google and Facebook is simply incompatible with the method of ensuring privacy presented in this thesis. Those two companies - and many other like them - virtually strip-mine user data to try and discover all possible connections to anything that someone might want to sell. If they do not have access to such data, then the scope of any data mining they might wish to do is drastically reduced.

Even if the data mining was moved to the client - and that is actually an interesting possibility worth investigating in its own right - they would not be able to pull in advertisements based on that information, as our model would detect that as violating the user's privacy. The only way I can see to work around this is to pull in the advertisements independently of user data, and then combine the two on the client. However, that solution does not scale well.

This is not to say that there are not *other* models of targeted advertising which could work, or other business models that could profit under these constraints. Just that this one does not.

It also bears mentioning that service critical data is, and always will be available to the service - to take an obvious example, Google will always know what you search for using its search engine, because it has to know in order to give you the results. There is nothing to stop a service from performing data mining on service critical data, other than the users choosing not to use that service.

Business models for which user privacy is a strength and/or a selling point obviously stand to benefit, as the work presented here provides not only a generic technique for providing user privacy, but also for allowing the user to ensure it.

8.2.3 Applicability to Service Models

The overall model presented in this thesis is not applicable to all web-services. Broadly speaking, it can be applied to two classes of web-service:

- Those that store sensitive user data for later retrieval by the user.
- Those that enable the sharing of sensitive data between users.

Social networks, and file storage are the two most notable classes of web-service to which the model applies.

It expressly does not apply to any service which requires all the information it collects in order to operate - i.e. a service in which all data is service-critical. Search engines are a notable instance of this.

Email is an interesting case - in principle, the model can be applied to it. However, it requires that both the sending and receiving email servers are capable of handling the required key exchange. In the event that the receiving server does not possess the capability, the sending server must choose between sending the email as plain text - which clearly does not protect the users' privacy - or sending it in an encrypted format, and foisting the problem of how to decrypt it on to the users.

In the event that the sending server does not possess the capability, the receiving server should encrypt the data upon arrival using the appropriate user's public key. This method is sufficient to protect against compromise of the data store, though not against compromise of the service, or against legal compromise.

The same principle applies to any "meta-service" which operates across the boundaries of individual services. Such services can in principle interoperate, but they require a way to perform key exchange with each other.

8.2.4 Cross-Site Scripting Attacks

Because the service is not capable of examining encrypted information, there exists the possibility that other users can launch cross-site scripting attacks by embedding malicious code into information they send to other users via the service.

This can be dealt with, but doing so requires that information from other users be parsed and checked for such vulnerabilities before it is added to the DOM. These checks must be performed on the receiving side. They cannot reliably be performed on the sending side, because the sender could disable them.

8.3 Implications for Research

8.3.1 Moving Computation to the Client

Because the service is no longer capable of performing computations over much of the user data, it is necessary to move at least some of these computations to the client. This may seem relatively simple, but it raises interesting points about how one should go about performing tasks such as indexing without the help of a MapReduce implementation, or search without the the assistance of supporting database software. JavaScript is hardly known for its speed, so performance could very easily become an issue with computationally heavy operations.

Additionally, techniques may need to be developed which allow for incremental updates to the data-structures backing such operations, rather than doing a full re-computation each time there is an alteration.

8.3.2 Homomorphic Encryption

An alternative to moving computation to the client is to utilise the concept of homomorphic encryption [48] to perform computations over the encrypted data, while it is still encrypted. This is a concept used by Mylar [35], in order to perform an encrypted search, which is a keyword search over multiple encrypted documents on the server.

To my knowledge[3], the general issues with homomorphic encryption are that it is many orders of magnitude slower than the equivalent computation over unencrypted data, and that it is difficult to use it for arbitrary computations. If the concept is to achieve widespread uptake - and thus assist in ensuring the privacy of the broader internet community - it will need to overcome such barriers to its use.

[3] And I am *really* not an expert on the matter.

Chapter 9

Conclusions

In this paper, I have presented a model under which genuine privacy can be delivered to the general population of internet users, in a way which is very close to being as usable as the services that they are currently used to using. This model protects against a very broad range of threats to user privacy, while simultaneously protecting the services operating it against the consequences of large-scale security breaches and data theft.

Furthermore, the privacy offered under the model is automatically verifiable, and a method for performing this verification was presented. While the details of this method still require work, they provide a means for users to have genuine peace of mind with regards to their privacy online.

Chapter 10

Acknowledgements

Many thanks to my supervisor, Roger Clarke, for his many helpful points of advice, for sticking with me and this project, and keeping me on track (or at least trying to), despite my not always being the best student.

Also to my partner, Claire, who kept me from going insane while doing this, and who somehow hasn't torn my head off, despite the fact that she's *also* trying to write a thesis.

And to Afnan Hannan and Jae Lee, with whom I worked on a rather interesting project, which sewed the seeds for this one.

Bibliography

[1] Alfred V Aho, Monica S Lam, Ravi Sethi, and Jeffrey D Ullman. *Compilers: principles, techniques, & tools*, volume 1009. Pearson/Addison Wesley, 2007.

[2] The OpenPGP Alliance. OpenPGP. `http://www.openpgp.org/`, 1991. [Online; accessed 2014-05-21].

[3] Various Authors. Certificate authority. `http://en.wikipedia.org/wiki/Certificate_authority`. [Online; accessed 2014-05-28].

[4] Various Authors. Transport layer security. `http://en.wikipedia.org/wiki/Secure_Sockets_Layer`. [Online; accessed 2014-05-28].

[5] Andrew Bloomgarden and NH Hanover. A data flow tracker and reference monitor for webkit and javascriptcore. 2012.

[6] Daniel Castro. How much will prism cost the u.s. cloud computing industry? `http://www2.itif.org/2013-cloud-computing-costs.pdf`, 2013. [Online; accessed 2014-05-03].

[7] World Wide Web Consortium. Web cryptography api. `http://www.w3.org/TR/WebCryptoAPI/`, 2014. [Online; accessed 2014-05-03].

[8] Colorado State Department. A guide to protecting your business and recovering from business identity theft. `http://www.sos.state.co.us/pubs/business/ProtectYourBusiness/BITresourceguide.html`, 2012. [Online; accessed 2014-05-21].

[9] Mozilla Developer Network and individual contributors. Ajax. `https://developer.mozilla.org/en/docs/AJAX`, 2014. [Online; accessed 2014-05-28].

[10] W. Diffie and M.E. Hellman. New directions in cryptography. *Information Theory, IEEE Transactions on*, 22(6):644–654, Nov 1976.

[11] Pietro Ferrara, Marco Pistoia, Omer Tripp, et al. LaBaSec: Language-based Security. `http://researcher.watson.ibm.com/researcher/view_group_subpage.php?id=1598`. [Online; accessed 2014-05-28].

[12] David Flannagan. Javascript: The definitive guide, 1997.

[13] United States District Court for the Eastern District of Virginia. Exhibit 9, attachment b. `https://s3.amazonaws.com/s3.documentcloud.org/documents/801182/redacted-pleadings-exhibits-1-23.pdf`, 2013. [Online; accessed 2014-05-03].

[14] Glen Greenwald and Ewen Macaskill. NSA Prism program taps in to user data of Apple, Google and others. `http://www.theguardian.com/world/2013/jun/06/us-tech-giants-nsa-data`, June 2013. [Online; accessed 2014-05-21].

[15] Alexa Internet Inc. Alexa Top 500 Global Sites. http://www.alexa.com/topsites/global, 2014. [Online; accessed 2014-05-21].

[16] Network Associates Inc. How pgp works. http://www.pgpi.org/doc/pgpintro/#p20, 1990-1999. [Online; accessed 2014-04-11].

[17] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities (technical report).

[18] The jQuery Foundation. jquery. http://jquery.com/, 2014. [Online; accessed 2014-05-28].

[19] Nadim Kobeissi. Thoughts on critiques of javascript cryptography. http://nadim.computer/post/84140092153/thoughts-on-critiques-of-javascript-cryptography, 2011-2013. [Online; accessed 2014-05-03].

[20] Nadim Kobeissi et al. Cryptocat. https://crypto.cat/, 2012. [Online; accessed 2014-05-21].

[21] Ladar Levison. Lavabit. http://lavabit.com/, 2013-2014. [Online; accessed 2014-05-28].

[22] Mega Limited. Help centre - security & privacy. https://mega.co.nz/#help_security, 2014. [Online; accessed 2014-05-03].

[23] Mega Limited. Mega. https://mega.co.nz/, 2014. [Online; accessed 2014-05-03].

[24] Mega Limited. Mega developer documentation. https://mega.co.nz/#doc, 2014. [Online; accessed 2014-05-03].

[25] Alfred J Menezes, Paul C Van Oorschot, and Scott A Vanstone. *Handbook of applied cryptography*. CRC press, 1996.

[26] Riley Michael, Ben Elgin, Dune Lawrence, and Carol Matlack. Missed alarms and 40 million stolen credit card numbers: How target blew it. http://www.businessweek.com/articles/2014-03-13/target-missed-alarms-in-epic-hack-of-credit-card-data, March 2014. [Online; accessed 2014-05-21].

[27] Mozilla Developer Network. Web api interfaces. https://developer.mozilla.org/en-US/docs/Web/API, 2005-2014. [Online; accessed 2014-04-23].

[28] Mozilla Developer Network and individual contributors. Debugger API. https://developer.mozilla.org/en-US/docs/Tools/Debugger-API, 2005-2014. [Online; accessed 2014-05-28].

[29] Mozilla Developer Network and individual contributors. Event. https://developer.mozilla.org/en/docs/Web/API/Event, 2005-2014. [Online; accessed 2014-05-28].

[30] Mozilla Developer Network and individual contributors. Navigatorplugins.plugins. https://developer.mozilla.org/en-US/docs/Web/API/NavigatorPlugins.plugins, 2005-2014. [Online; accessed 2014-05-28].

[31] Mozilla Developer Network and individual contributors. try...catch. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/try...catch, 2005-2014. [Online; accessed 2014-05-28].

[32] Mozilla Developer Network and individual contributors. window.crypto.getrandomvalues. http://www.w3.org/TR/WebCryptoAPI/, 2005-2014. [Online; accessed 2014-05-03].

[33] Mozilla Developer Network and individual contributors. Worker. https://developer.mozilla.org/en-US/docs/Web/API/Worker, 2005-2014. [Online; accessed 2014-05-28].

[34] Oracle. Manipulating dom of applet's web page. http://docs.oracle.com/javase/tutorial/deployment/applet/manipulatingDOMFromApplet.html, 1995-2014. [Online; accessed 2014-05-28].

[35] Raluca Ada Popa, Emily Stark, Steven Valdez, Jonas Helfer, Nickolai Zeldovich, and Hari Balakrishnan. Building web applications on top of encrypted data using mylar. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 157–172, Seattle, WA, April 2014. USENIX Association.

[36] Kevin Poulsen. Edward snowden's e-mail provider defied fbi demands to turn over crypto keys, documents show. http://www.wired.com/2013/10/lavabit_unsealed/, 2013. [Online; accessed 2014-05-03].

[37] The Tor Project. Tor project. https://www.torproject.org/. [Online; accessed 2014-05-28].

[38] Ronald L Rivest, Adi Shamir, and Len Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[39] Robusto et al. Can an embedded flash object access the dom of its parent document? http://stackoverflow.com/a/2473061/3375042, 2010. [Online; accessed 2014-05-28].

[40] Andrei Sabelfeld and Andrew C Myers. Language-based information-flow security. *Selected Areas in Communications, IEEE Journal on*, 21(1):5–19, 2003.

[41] Aaron Miller Paramjit Singh Sandhu. Taintsniffer: A robust dynamic taint tracking system for a homogenous web browsing environment.

[42] Golam Sarwar, Olivier Mehani, Roksana Boreli, and Dali Kaafar. On the effectiveness of dynamic taint analysis for protecting against private information leaks on android-based devices. In *10th International Conference on Security and Cryptography (SECRYPT)*, 2013.

[43] Bruce Schneier. Description of a new variable-length key, 64-bit block cipher (blowfish). In *Fast Software Encryption*, pages 191–204. Springer, 1994.

[44] Matasano Security. Javascript cryptography considered harmful. http://www.matasano.com/articles/javascript-cryptography/, 2011. [Online; accessed 2014-05-03].

[45] Richmond Shane and Christopher Williams. Millions of internet users hit by massive sony playstation data theft. http://www.telegraph.co.uk/technology/news/8475728/

Millions-of-internet-users-hit-by-massive-Sony-PlayStation-data-theft.html, April 2011. [Online; accessed 2014-05-21].

[46] Saurabh Sinha and Mary Jean Harrold. Analysis of programs with exception-handling constructs. In *Software Maintenance, 1998. Proceedings., International Conference on*, pages 348–357. IEEE, 1998.

[47] NIST-FIPS Standard. Announcing the advanced encryption standard (aes). *Federal Information Processing Standards Publication*, 197, 2001.

[48] Craig Stuntz. What is homomorphic encryption, and why should i care? http://blogs.teamb.com/craigstuntz/2010/03/18/38566/, 2010. [Online; accessed 2014-05-28].

[49] Philipp Vogt, Florian Nentwich, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Cross site scripting prevention with dynamic data tainting and static analysis. In *NDSS*, 2007.

[50] W3Schools. Navigator taintEnabled() method. http://www.w3schools.com/jsref/met_nav_taintenabled.asp, 2014. [Online; accessed 2014-05-21].

[51] Shiyi Wei and Barbara G Ryder. Practical blended taint analysis for javascript. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, pages 336–346. ACM, 2013.

[52] Mari Yamaguchi. Sony playstation network hack to cost $170 million. http://www.huffingtonpost.com/2011/05/23/sony-playstation-network-hack-cost_n_865432.html, 2011. [Online; accessed 2014-05-21].

[53] Andy Yen, Jason Stockman, Wei Sun, et al. Protonmail. https://protonmail.ch/, 2014. [Online; accessed 2014-05-21].