SHPF: Enhancing HTTP(S) Session Security with Browser Fingerprinting (extended preprint)

Thomas Unger
FH Campus Wien
Vienna, Austria

Martin Mulazzani, Dominik Frühwirt,
Markus Huber, Sebastian Schrittwieser, Edgar Weippl
SBA Research
Vienna, Austria
Email: (1stletterfirstname)(lastname)@sba-research.org

***Abstract*—Session hijacking has become a major problem in today's Web services, especially with the availability of free off-the-shelf tools. As major websites like Facebook, Youtube and Yahoo still do not use HTTPS for all users by default, new methods are needed to protect the users' sessions if session tokens are transmitted in the clear.**
In this paper we propose the use of browser fingerprinting for enhancing current state-of-the-art HTTP(S) session management. Monitoring a wide set of features of the user's current browser makes session hijacking detectable at the server and raises the bar for attackers considerably. This paper furthermore identifies HTML5 and CSS features that can be used for browser fingerprinting and to identify or verify a browser without the need to rely on the UserAgent string. We implemented our approach in a framework that is highly configurable and can be added to existing Web applications and server-side session management with ease. To enhance Web session security, we use baseline monitoring of basic HTTP primitives such as the IP address and UserAgent string, as well as complex fingerprinting methods like CSS or HTML5 fingerprinting. Our framework can be used with HTTP and HTTPS alike, with low configurational and computational overhead. In addition to our contributions regarding browser fingerprinting, we extended and implemented previous work regarding session-based shared secrets between client and server in our framework.

***Keywords*-Session Hijacking, Browser Fingerprinting, Security**

I. Introduction

Social networks and personalized online services have an enormous daily user base. However, Internet users are constantly at risk. Popular websites like Facebook or Yahoo, along with many others, use HTTPS-secured communication only for user authentication, while the rest of the session is usually transmitted in the clear. This allows an attacker to steal or copy the session cookies, identifiers or tokens, and to take over the session of the victim. Unencrypted Wi-Fi and nation-wide interceptors have used this as an attack vector multiple times recently, proving that session hijacking is indeed a problem for today's Internet security. A recent prominent example includes the hacked Twitter account of Ashton Kutcher [20], who used an unencrypted Wi-Fi and got hacked— at that time, he had more than six million followers. Tunisia on the other hand was accused of malicious JavaScript injection on websites like Facebook, Gmail and Yahoo, to harvest login credentials and sabotage dissidents online activities [11].

To protect the session of a user, we implemented a framework that ties the session to the current browser by fingerprinting and monitoring the underlying browser, its capabilities, and detecting browser changes at the server side. Our framework, the *Session Hijacking Prevention Framework (SHPF)*, offers a set of multiple detection mechanisms which can be used independently of each other. SHPF protects especially against session hijacking of local adversaries, as well as against cross-site scripting (XSS). The underlying idea of our novel framework: If the user's browser suddenly changes from, e.g., Firefox on Windows 7 64 bit to an Android 4-based Webkit browser in a totally different IP range, we assume that some form of mischief is happening.

Our framework uses a diverse set of inputs and allows the website administrator to add SHPF with just a few additional lines of code in existing applications. There is no need to change the underlying Web application, and we can use the initial authentication process which is already part of many applications to build further security measurements on top. As part of the authentication process at the beginning of a session, the server asks the browser for an exact set of features and then monitors constantly whether the browser still behaves as expected over the entire session. While an attacker can easily steal unencrypted session information, e.g., on unencrypted Wi-Fi, it is hard to identify the exact responses needed to satisfy the server without access to the same exact browser version. Furthermore, we use a shared secret that is negotiated during the authentication, which is used to sign requests with an HMAC and a timestamp, building and improving on previous work in this direction. Recent attacks against HTTPS in general and the certificate authorities Diginotar and Comodo [22] in particular have shown that even the widespread use of SSL and HTTPS are not sufficient to protect against active adversaries and session hijacking. Previous work in the area of server-side session hijacking prevention relied, e.g., on a shared secret that is only known to the client's browser [1] and never transmitted in the clear. While this is a feasible approach and can protect a session even for unencrypted connections, our system extends this method by employing browser fingerprinting for session security, thus allowing us to incorporate and build upon existing security mechanisms like HTTPS. Furthermore, it offers

protection against both passive and active adversaries.

Our contributions in this paper are the following:

- We present a framework to enhance HTTP(S) session management, based on browser fingerprinting.
- We propose new browser fingerprinting methods for reliable browser identification based on CSS3 and HTML5.
- We extend and improve upon existing work on using a shared secret between client and server per session.
- We have implemented the framework and will release the code and our test data under an open source license[1].

The rest of the paper is organized as follows: Section II gives a brief technical background. The new browser fingerprinting methods are presented in Section III. Our SHPF framework and its general architecture is described in Section IV. We evaluate our framework in Section V. The results of our evaluation are discussed in Section VI, before we conclude in Section VII.

II. Background

Web browsers are very complex software systems requiring multiple person-years of development time. Different international standards like HTML, JavaScript, DOM, XML or CSS specified by the W3C[2] try to make the browsing experience across different browsers as uniform as possible, but browsers still have their own "touch" in interpreting these standards - a problem Web developers have been struggling with since the infancy of the Web. Due to the complexity of the many standards involved, there are differences in the implementations across browsers. New and upcoming standards further complicate the landscape - HTML5 and CSS3 for example, which are not yet fully standardized but already partly implemented in browsers. These imperfect implementations of standards with different depth are perfectly suited for fingerprinting. *Nmap* [24], for example, uses this exact methodology to identify the operating system used on a remote host based on TCP/IP stack fingerprinting.

Authentication on websites works as follows: A HTML form is presented to the user, allowing them to enter username and password, which are then transmitted to the server. If the login succeeds, the server typically returns a token (often referred to as a session ID), which is subsequently sent along with further client requests to identify the user due to the stateless internals of the HTTP protocol. In an unencrypted HTTP environment, this presents multiple challenges to the user's confidentiality: If login credentials are transmitted in an unencrypted state, an eavesdropping attacker can learn them without any further effort. Even if the document containing the login form as well as the subsequent request containing the credentials are transmitted over HTTPS, attackers may later learn the session ID from unencrypted requests to the server or by using client-side attacks such as XSS. Thus, it is imperative to enforce SSL throughout the entire site (or at least on those domains that are privileged to receive the session token).

Many administrators regard introducing SSL by default as too cost-intensive. Anecdotal evidence suggests that naively enabling SSL without further configuration may incur significant performance degradation up to an order of magnitude. Gmail, however, switched to HTTPS by default in January 2010 [21]. Remarkably, Google reported that they did not deploy any additional machines and no special hardware (such as hardware SSL accelerators), but employed a number of SSL optimization methods. Only about 10 kB memory per connection, 1% of the CPU load and less than 2% network overhead were incurred for SSL in this configuration. Many other problems with HTTPS have been discussed in the literature, ranging from problems with the CA system [36], [8] to the fact that a large number of keys have been created with weak overall security [12], [23], [44]. While HTTPS can be found on a large number of popular websites that require some form of authentication [13], only a minority of these binds the sessions to a user's device or IP address to protect the user against session hijacking [3].

Multiple tools have been released that allow automated session hijacking: FaceNiff [34], DroidSheep[3], Firesheep, cookiemonster [32] and sslstrip, just to name a few. Firesheep was among the first to received widespread public attention when it was released as open source in 2010 [4]. Firesheep works as follows: Upon startup, Firesheep tries to start sniffing on an IEEE 802.11 or Ethernet device. Whenever HTTP packets are captured and can be parsed as such, they are matched against domain-specific handlers (as of writing, the current git repository includes handlers for sites such as Facebook, Google, and LinkedIn). These handlers store a list of cookie values that comprise a valid session. When a match is found, these values are extracted and an entry for this hijackable session is added to the attacker's sidebar in Firefox. When the attacker selects one of these entries, Firesheep writes the stored cookie values into Firefox' cookie manager and opens the site in a new tab, thereby presenting the attacker with a hijacked and fully operational session of the victim.

III. Browser Fingerprinting

This section introduces our new browser fingerprinting methods, namely CSS and HTML5 fingerprinting. While browser fingerprinting has ambiguous meanings in the literature i.e., identifying the web browser down to the browser family and version number [7] vs. (re-)identifing a given user [26], we use the former. Our framework relies on fingerprinting to reliably identify a given browser, and CSS fingerprinting is one of the fingerprinting techniques

[1]Note to the reviewer: we will include the link here once the paper is accepted for publication

[2]http://www.w3.org/TR/

[3]http://droidsheep.de/

Browser	Layout Engine	Prefix
Firefox	Gecko	-moz-
Konqueror	KHTML	-khtml-
Opera	Presto	-o-
Internet Explorer	Trident	-ms-
Safari	Webkit	-webkit-
Chrome	Webkit	-webkit-

Table I
BROWSER LAYOUT ENGINES AND CSS PREFIXES

implemented in our framework. Furthermore, we present details on how we monitor HTTP headers in SHPF, which allows website administrators to configure advanced policies such as preventing an HTTP session from roaming between a tightly secured internal network and a public network beyond the control of the administrators.

A. CSS Fingerprinting

CSS as a standard is under ongoing development and standardization. CSS 2.1 was published as a W3C Recommendation in June 2011, while the upcoming CSS3 is not yet finished. The CSS3 modules vary in stability and status and while some of them already have recommendation status, others are still candidate recommendations or working drafts. Browser vendors usually start implementing properties early, even long before they become recommendations. We identify three CSS-based methods of browser fingerprinting: CSS properties, CSS selectors and CSS filters.

Depending on the layout engine, progress in implementation varies for new and upcoming CSS properties, which allows us to identify a given browser by the CSS properties it supports. Table I shows which browser uses which layout engine. When properties are not yet on ”Recommendation“ or ”Candidate Recommendation“ status, browsers prepend a vendor-specific prefix indicating that the property is supported for this browser type only. Table I also shows the vendor prefixes for the most popular browsers. Once a property moves to Recommendation status, prefixes are dropped by browser vendors and only the property name remains. For example, in Firefox 3.6 the property *border-radius* had a Firefox prefix resulting in *-moz-border-radius*, while in Chrome 4.0 and Safari 4.0 it was *-webkit-border-radius* (as they both use the Webkit layout engine). Since Firefox 4 as well as Safari 5.0 and Chrome 5.0 this feature is uniformly implemented as *border-radius*. The website https://www.caniuse.com shows a very good overview on how CSS properties are supported in the different browsers and their layout engine.

Apart from CSS properties, browsers may differ in supported CSS selectors as well. Selectors are a way of selecting specific elements in an HTML tree. For example, CSS3 introduced new selectors for old properties, and they too are not yet uniformly implemented and can be used for browser fingerprinting.

The third method of distinguishing browsers by their behavior is based on CSS filters. CSS filters are used to modify the rendering of e.g., a basic DOM element, image, or video by exploiting bugs or quirks in CSS handling for specific browsers, which again is very suitable for browser fingerprinting. Centricle [4] provides a good comparison of CSS filters across different browsers.

How to test: As CSS is used for styling websites it is difficult to compare rendered websites at the server side. Instead of conducting image comparison (as used recently by Mowery et al. [29] to fingerprint browsers based on WebGL-rendering), we use JavaScript in our implementation to test for CSS properties in style objects: in DOM, each element can have a style child object that contains properties for each possible CSS property and its value (if defined). These properties in the style object have the same name as the CSS property, with a few differences, for example dashes (-) are removed, the following letter becomes upper case. Vendor-specific prefixes however are preserved if the CSS property has a vendor prefix. An Example: *-moz-border-radius* becomes *MozBorderRadius*.

There are now two ways to test CSS support of a property in the style object: the first way is to simply test whether the browser supports a specific property by using the *in* keyword on an arbitrary style object. The returning Boolean value indicates whether the property is supported. Browser-specific prefixes need to be considered when testing properties with this method. An example: *'borderRadius' in document.body.style*. The second way to test whether a given CSS property is supported is to look at the value of a property once it has been set. We can set an arbitrary CSS property on an element and query the JavaScript *style* object afterwards. Interpreting the return values shows whether the CSS property is supported by the browser: *undefined (null)* as return value indicates that the property is not supported. If a not-*null* value is returned this means the property is supported and has been parsed successfully by the browser.

Care has to be taken when interpreting the return values for fingerprinting: A returning value may deviate from the CSS definition if some parts were not understood by the browser. This can happen, e.g., with composite properties, which allow several sub-properties to be defined in just one long definition. For example, the *background* definition can be used to define *background-color, background-repeat, background-image, background-position* and *background-attachment* all at once. Interestingly, the value string returned upon querying the style object also differs between browsers, and can be used as yet another test for fingerprinting based on CSS properties. For example, consider the following CSS3 background

[4]http://centricle.com/ref/css/filters/

definition:

background:hsla(56, 100%, 50%, 0.3)

Upon testing the background property on the style object as described above, Firefox returns the following:

none repeat scroll 0% 0% rgba(255, 238, 0, 0.3)

Internet Explorer, on the other hand, returns this:

hsla(56, 100%, 50%, 0.3)

As this shows, Firefox returns all possible values of the composite background property explained above (repeat, color, image, position) and additionally converts the *hsla* definition to *rgba* values. In contrast, Internet Explorer only returns exactly what was stated in the CSS definition, no more and no less, and does not convert the values into another format. The order of elements within the return string for composite values may also deviate between browsers, for example with the *box-shadow* property with distance values as well as color definitions.

B. HTML5 Fingerprinting

HTML5, like CSS3, is still under development, but there are already working drafts which have been implemented to a large extend by different browsers. This new standard introduces some new tags, but also a wide range of new attributes. Furthermore HTML5 specifies new APIs (application programming interfaces), enabling the Web designer to use functionalities like drag and drop within websites. Since browser vendors have differing implementation states of the new HTML5 features, support for the various improvements can be tested and used for fingerprinting purposes as well. For identifying the new features and to what extent they are supported by modern browsers, we used the methodology described in [33]. The W3C furthermore has a working draft on differences between HTML5 and HTML4 that was used as input [38].

In total we identified a set of 242 new tags, attributes and features in HTML5 that were suitable for browser identification. While 30 of these are attributed to new HTML tags that are introduced with HTML5 [41], the rest of the new features consist of new attributes for existing tags as well as new features. We then created a website using the Modernizr [2] library to test for each of these tags and attributes and whether they are supported by a given browser. We collected the output from close to 60 different browser versions on different operating systems. An excerpt of the data and how the tags and attributes are supported by different browsers can be seen in Table II. One of our findings from the fingerprint collection was that the operating system apparently has no influence on HTML5 support. We were unable to find any differences between operating systems while using the same browser version, even with different architectures. An example: Firefox 11 on Windows XP (32 bit) and on Windows 7 (64 bit) share the same fingerprint.

C. Basic HTTP Header Monitoring

For each HTTP request, a number of HTTP headers is included and transmitted to the Web server. RFC 2616 defines the HTTP protocol [10] and specifies several HTTP headers that can or should be sent as part of each HTTP request. The number of headers, the contents and especially the order of the header fields, however, are chosen by the browser and are sufficient for identifying a browser. Using this method for browser identification has already been discussed in previous work [7], [43] and is already used to some extend by major websites [3], we will thus only briefly cover the parts which are of particular interest for SHPF. In our implementation we use the following header fields for HTTP session monitoring:

- **UserAgent string** contains browser version and platform information.
- **Accept** specifies which data types the browser supports. It is also used to announce a preference for a certain data type.
- **Accept-Language** specifies, similar to Accept, which language is preferred by the browser.
- **Accept-Encoding** specifies which encodings are supported and which encoding is preferred by the browser.
- **IP-Address** of the client is not part of the HTTP header. However, the client IP address can be processed by the server easily.

The UserAgent contains information about the browser - often the exact browser version and the underlying operating system. It is, however, not a security feature, and can be changed arbitrarily by the user. SHPF is not dependending on the UserAgent, and works with any string value provided by the browser. If the UserAgent changes during a session this is a strong indication for session hijacking, especially across different web browsers. Depending on the configuration of SHPF and the particular security policy in place, it might however be acceptable to allow changes in the browser version e.g., with background updates of the browser while using a persistent session if the browser is restarted.

The UserAgent as well as the other headers and data usually remain consistent during a session. If any values or a subset of these values change during a session, the session has been hijacked (in the simplest case). For example, if during a session multiple UserAgents from different IPs use the same session cookie, this implies in our framework that the session has been hijacked (session identifiers ought to be unique). The session would be terminated immediately and the user would need to reauthenticate. In order to bypass this part of our framework, the attacker would need to replicate all the original headers and use the same IP address as the client in order to send valid requests to the server. While cloning the HTTP headers is rather easy, binding a session to a given IP address considerably raises the bar for adversaries, even if they can obtain a valid session cookie and the HTTP header with e.g., one of various

Tag	Attribute	FF12	FF13	C18	C19	IE8	IE9	O11	O12	S4	S5
<audio>	—	✓	✓	✓	✓	✗	✓	✓	✓	✗	✗
<fieldset>	name	✓	✓	✗	✓	✗	✗	✗	✗	✗	✗
<textarea>	maxlength	✓	✓	✓	✓	✗	✗	✓	✓	✗	✓
<nav>	—	✓	✓	✓	✓	✗	✓	✗	✓	✗	✓
<meter>	—	✗	✗	✓	✓	✗	✗	✓	✓	✗	✗
<input>	type="url"	✓	✓	✓	✓	✗	✗	✓	✓	✗	✓
<canvas>	—	✓	✓	✓	✓	✗	✓	✓	✓	✓	✓

Table II
EXCERPT OF HTML5 TAGS AND ATTRIBUTES FOR BROWSER FINGERPRINTING

kinds of cross-site scripting (XSS) attack [37].

Apart from the HTTP header values themselves, there is also a significant difference in how the browsers order the HTTP header fields. While Internet Explorer 9 for example sends the UserAgent before the Proxy-Connection and Host header fields, Chrome sends them in the exact opposite order. The content of the header fields is not important in this case, all header fields are included for this check in our implementation. HTTP header ordering is especially useful against session hijacking tools like that clone only the UserAgent or copy the session cookie, but not the rest of the header information.

IV. SHPF FRAMEWORK

This section describes the implementation of our framework and its architecture, the *Session Hijacking Prevention Framework (SHPF)*. The source code is released under an open source license and can be found on github[5]. Despite the new fingerprinting methods presented in the previous section, we also implemented and improved SessionLock [1] for environments that do not use HTTPS by default for all connections.

A. General Architecture

SHPF is a server-side framework which is written in PHP5 and consists of multiple classes that can be loaded independently. Its general architecture and basic functionality is shown in Figure 1. We designed it to be easily configurable (depending on the context and the security needs of the website), portable and able to handle a possibly large number of concurrent sessions. Our implementation can be easily extended with existing and future fingerprinting methods, e.g., textfont rendering [29] or JavaScript engine fingerprinting [28], [35].

The main parts of the framework are the so-called *features*. A feature is a combination of different checks for detecting and mitigating session hijacking. In our prototype we implemented the following features: HTTP header monitoring, CSS fingerprinting and SecureSession (which implements and extends the SessionLock protocol by Ben Adida). Features are also the means to extending the framework, and we provide base classes for fast feature development. A feature consists of one or more *checkers*, which are used to run certain tests. There are two different types (or classes) of checkers:

- *Synchronous checkers* can be used if the tests included in the checker can be run solely from existing data, such as HTTP requests or other website-specific data that is already available.
- *Asynchronous checkers* are used if the tests have to actively request some additional data from the client and the framework has to process the response.

While synchronous checkers are passive in nature, active checkers can challenge the client to send some information for a specific checker, allowing the server to verify that the browser is behaving as expected. Client responses are sent via asynchronous calls (AJAX) as part of SHPF, thus not blocking the session or requiring to rewrite any existing code. Appendix A shows the basic code needed to incorporate SHPF into a website.

The distinction between features and checkers gives the website control over which checks to run. Features can be disabled or enabled according to the website's security needs, and can be assigned to different security levels. Different security levels within a webpage are useful, for example, in privacy-heterogeneous sessions - basic checks are performed constantly, while additional checks can be run only when necessary, e.g., when changing sensitive information in a user's profile (much like Amazon does for its custom session management). In order to communicate with a Web application, callbacks can be defined both in PHP and JavaScript. These callbacks are called if a checker fails and thus allow the application to react in an appropriate way, e.g., terminate the session and notify the user. An example configuration for different security levels with SHPF can be seen in Table III. The details for each checker in this example are explained in detail below. Consider a website, e.g., a web store, which uses three different security levels for every session:

- Level 1 is for customers who are logged in and browsing the web store.
- Level 2 is for customers who are in a sensitive part of their session, e.g., ordering something or changing their profile.
- Level 3 is for administrators who are logged into the administrative interface.

Level 1 is a very basic security level. In this example it

[5]https://github.com/mmulazzani/SHPF

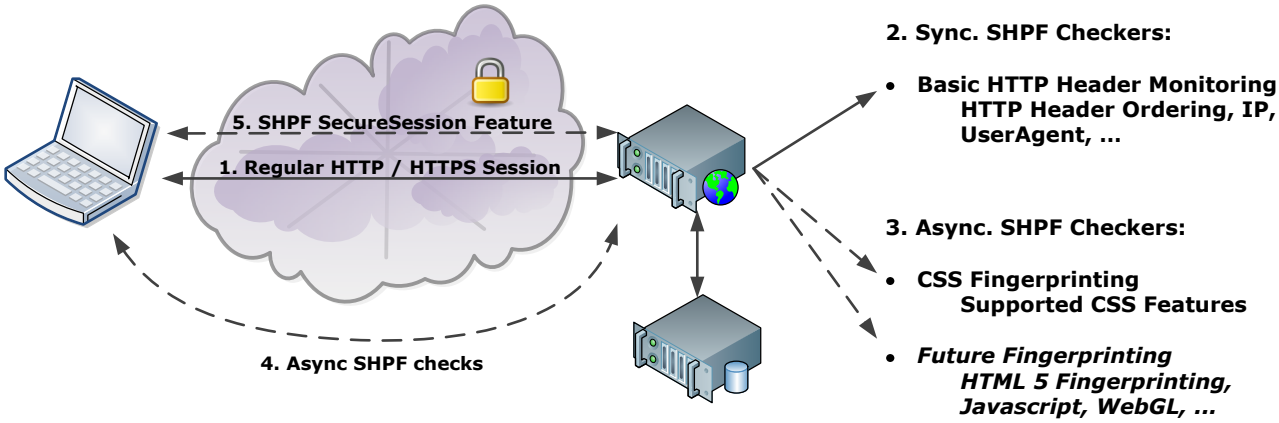

Figure 1. SHPF Architecture

prevents session hijacking by monitoring the UserAgent string of the user for modifications. As a sole security measure it only protects the user against attacks that can be considered a nuisance, and can possibly be bypassed by an attacker (by cloning the UserAgent string). The Web application is designed in such a way that an attacker cannot actively do any harm to the user, for example browsing only specific products to manipulate the web store's recommendation fields. If the customer decides to buy something, level 2 is entered, which uses two additional security measures: the current session is locked to the user's IP address and the order of the HTTP headers is monitored to detect if a different browser uses the same UserAgent string. Once the transaction is complete, the customer returns to level 1. For an administrator, even more checkers are enabled at the start of the session: SecureSession protects the session cryptographically with a shared secret between the particular browsers that started the sessions, and the CSS properties supported by the browser are monitored. Please note that this configuration is given merely by way of an example and must be matched to existing security policies when implemented. Furthermore, note that HTTPS is not mentioned in the example - even though it is strongly advised to use HTTPS during a session (besides SHPF), it is not a requirement. SHPF can prevent session hijacking even if session tokens are transmitted in the clear.

Checks	Security Levels		
	Level 1	Level 2	Level 3
UserAgent monitoring	✓	✓	✓
IP binding	✗	✓	✓
HTTP Header ordering	✗	✓	✓
CSS fingerprinting	✗	✗	✓
SecureSession	✗	✗	✓

Table III
EXAMPLE - DIFFERENT SECURITY LEVELS FOR A WEBSITE

Additional components in our framework are used for keeping track of the session state in a database, for output and logging, and there is a special class for integrating the framework into existing websites and a crypto feature *CryptoProvider*. The crypto feature defines methods of encrypting and decrypting data. If a crypto provider is set in SHPF and SecureSession is used, all asynchronous messages exchanged between the browser and the framework are automatically encrypted by the crypto provider (see Section IV-D).

B. Basic HTTP Header Monitoring

The HTTP header monitoring feature does the following:

1) On the first request, the framework stores the contents and the order of the HTTP headers as described above.
2) For each subsequent request, the feature compares the headers sent by the client with the stored ones and checks whether their content and/or order match.

Depending on the particular use case, different configurations are possible, e.g., binding a session to a given IP, a certain IP range or a UserAgent string. Another example would be to allow IP address roaming while enforcing that the operating system as claimed by the UserAgent string as well as the browser has to stay the same, allowing the browser version to change, e.g., through background updates in Chrome or Firefox. HTTP header monitoring is implemented as a synchronous checker, as the data needed for processing is sent with every request.

C. CSS Fingerprinting

Using the CSS fingerprinting methods explained above, a SHPF feature has been implemented that does the following:

1) Check whether the client's browser supports JavaScript.
2) On the first request of the client: Run the complete fingerprinting suite on the client (using 23 CSS properties at the time of writing) and save the values.
3) For each subsequent request of the client: choose a subset of CSS properties and test them on the client.

4) Receive the data and check if it was requested by the framework (anti-replay protection).
5) Compare the values with the saved values.

As this feature needs data from the client, this checker has been implemented as an asynchronous checker. The client is challenged to answer a subset of the previously gathered properties either for each HTTP request or within a configurable interval between CSS checks (say, every 10 or 20 requests). By default, the framework tests three CSS properties and compares the results with the previously collected fingerprint of that browser. The data received asynchronously must match the requested properties and must arrive within a configurable time span. If the received data do not match the expected data, arrive too late or are not requested by the feature, the session is terminated immediately. If no response is received within a configurable time span, the session is terminated as well. SHPF may be also configured to terminate the session if no JavaScript is enabled, thus making CSS fingerprinting mandatory by policy.

In order to reliably identify a given browser, we selected a mix of CSS3 properties that are not uniformly supported by current browsers. In total, 23 properties were identified as suitable for fingerprinting. The website http://www.caniuse.com was used to identify CSS properties that are not uniformly compatible across browsers, as well as properties that still have vendor prefixes. The 23 identified properties, possible testing values, and their status in the standardization and implementation process are shown in Table IV. Please note that in some cases merely the additional values of an already existing property are new, while the CSS property itself is not a novel CSS feature.

For each CSS property, an empty HTML * * element is inserted into the page, which contains an inline CSS definition. The element is assigned an ID so that it can be later accessed with JavaScript. Such an element might, e.g., look like this:

* *

JavaScript is then used to check whether the properties set previously exist in the style object, and also to query the property's value. The answers from the client are collected in an array, which is then converted into JSON and sent to the server secured by HTTPS or the SecureSession feature against eavesdroppers.

["minWidth" in $("cssCheck1").style,$("cssCheck1").style.minWidth]

For our implementation of CSS fingerprinting in SHPF we chose to use CSS properties only - CSS selectors were not used because CSS properties are sufficient to reliably identify a given browser. Nonetheless, the framework could be extended by supporting CSS selector and CSS filter fingerprinting in the future.

D. SecureSession

The SecureSession feature implements the SessionLock protocol by Ben Adida [1], but extends and modifies it in certain aspects:

- SessionLock utilizes HTTPS to transfer the session secret to the client. In our SecureSession feature we use a Diffie-Hellman Key Exchange [5]as discussed by Ben Adida in his paper because of the recent attacks against the trust foundation of HTTPS (Diginotar, Comodo) to do the same. We also looked at performance and security implications of that choice.
- We use the new WebStorage [40] features implemented by modern browsers by using JavaScript and the *localStorage* object to store the session secret.
- We improved patching of URLs in JavaScript compared to the original protocol.

SessionLock used the URL Fragment Identifier to keep the session secret around but hidden in the network traffic. For that, each URL needs to be patched so that the fragment gets appended. Using WebStorage is superior in multiple ways. If WebStorage is not supported by the browser, SecureSession falls back to using the fragment identifier. SessionLock furthermore hooks into the XMLHttpRequest object to intercept and modify asynchronous messages. We determined that there are cross-browser issues in using this method. In order to improve compatibility across browsers, we used the modified XMLHttpRequest object by Sergey Ilinsky [16] to make message interception compatible across different browsers.

In order to implement the above features we used two checkers for the framework feature:

- The **SecureSessionSecretNegotiation**-Checker is an asynchronous checker. The server has to run a Diffie Hellman Key Exchange only if no valid session secret is present. The server first calculates its private and public parts, sends them to the client as JavaScript code, and receives the client parts asynchronously in response when the client is done calculating. Both sides can then calculate the shared session secret.
- The **SecureSessionSecretChecker**-Checker is a synchronous checker that validates all incoming requests regarding HMAC and timestamp.

The SecureSessionSecretNegotiation initiates the key exchange by sending JavaScript to the client containing the calculations as well as the prime, generator and public number. The client sends its public number back via an asynchronous call. The server assumes that JavaScript is disabled if it receives no answer from the client within a (configurable) period of time. Again, SHPF can be configured to make this feature mandatory. If an answer is received, all further requests need to be appended with a valid HMAC and timestamp (configurable). This is done by the SecureSessionSecretChecker. While the method is the same as in SessionLock, we ported it to PHP. However, there is an exception to the rule: As Ben Adida discussed in his paper, there may be cases where a URL is not valid, such as when a page is opened from a bookmark. In such a case, the feature allows a configurable amount of consecutive requests that may

CSS Status - Recommendation		CSS Status - Working Draft	
Feature	Value	Feature	Value
display	inline-block	transform	rotate(30deg)
min-width	35px	font-size	2rem
position	fixed	text-shadow	4px 4px 14px #969696
display	table-row	background	linear-gradient (left, red, blue 30%, green)
opacity	0.5	transition	background-color 2s linear 0.5s
background	hsla(56, 100%, 50%, 0.3)	animation	name 4s linear 1.5s infinite alternate none
CSS Status - Cand. Recommendation		resize	both
Feature	Value	box-orient	horizontal
box-sizing	border-box	transform-style	preserve-3d
border-radius	9px	font-feature-setting	dlig=1,ss01=1
box-shadow	inset 4px 4px 16px 10px #000	width	calc(25% -1em)
column-count	4	hyphens	auto
		object-fit	contain

Table IV
23 CSS PROPERTIES AND VALUES IDENTIFIED FOR CSS FINGERPRINTING

fail. If a valid request is received before that amount is exceeded, no action is taken. To make the key exchange secure against MITM attacks, this feature should only be used on top of HTTPS or a secure, offline communication channel for exchanging the parameters and the JavaScript code.

For implementation we used the *Crypt_DiffieHellman* library from the PEAR Framework[6] on the server side. On the client, we used the *Big Integer Library* of Leeom Baird[7]. The SecureSession feature also implements a CryptoProvider. The CryptoProvider offers AES-CBC encryption by using the SHA-1 hash of the session secret negotiated between the framework and the client as the key. For PHP, the PHP extension *mcrypt*[8] is used, for JavaScript we use the library *crypto-js*[9]. The CryptoProvider then encrypts all asynchronous messages from the client to the framework. Furthermore, it prepends a timestamp to the plaintext before encryption, thus preventing replay attacks if the age of the timestamp exceeds a configurable time span.

E. Further Fingerprinting Methods

Our framework is especially designed to allow new and possibly more sophisticated fingerprinting methods to be added at a later point in time by implementing them as additional checkers. The presented results on HTML5 fingerprinting above, e.g., have not yet been implemented at the time of writing. We are planning to implement HTML5 fingerprinting as an asynchronous checker in the near future. Other fingerprinting methods e.g., EFF's Panopticlick, can be added at ease adding 18.1 bits of entropy on average [7]. See Section VI-A for related work and other fingerprinting methods which could be added to SHPF.

V. EVALUATION

There are multiple possible attack vectors that enable an attacker to obtain session tokens of any kind and take over the victim's session. We will discuss for each attack vector how SHPF can detect session hijacking and how it prevents it.

A. Threat Model

An attacker in our threat model can be local or remote from the victim's point of view, as well as either active or passive. While a passive attacker just listens without interacting with the client, an active attacker sends, modifies or actively drops communication content. Different requirements have to be met for each of the outlined attacks, however, these are beyond the scope of this paper.

Figure 2 shows an overview of the different points of attack that were considered while designing SHPF. They are based on the OWASPS Top 10 from 2010[10], which has multiple categories that either directly allow session hijacking, or facilitate it. The most notable categories are "A2 Cross-Site Scripting", "A3 Broken User Authentication and Session Management" and "A9 Insufficient Transport Layer Protection". We particularly considered threats that are actively exploited in the wild, with tools available for free.

The following points of attack allow an attacker to hijack a session:

1) Different attacks where the attacker has access to the victim's network connection.
2) The target website is vulnerable to code injection attacks (XSS), pushing malicious code to the client.
3) Local code execution within the victims browser's sandbox, e.g., by tricking the victim into executing Javascript (besides XSS).
4) Attacker has access to 3rd party server with access to the session token, e.g., a proxy, Tor sniffing or via HTTP referrer string.

The detailed attack descriptions for each of these attacks are as follows: 1) If the attacker is on the same network as the victim, e.g., on unencrypted Wi-Fi, searching for unencrypted session tokens is trivial - these are the methods used, for example, by Firesheep and

[6]http://pear.php.net/package/Crypt_DiffieHellman
[7]http://leemon.com/crypto/BigInt.html
[8]http://php.net/manual/en/book.mcrypt.php
[9]https://code.google.com/p/crypto-js/
[10]https://owasp.org/index.php/Top_10

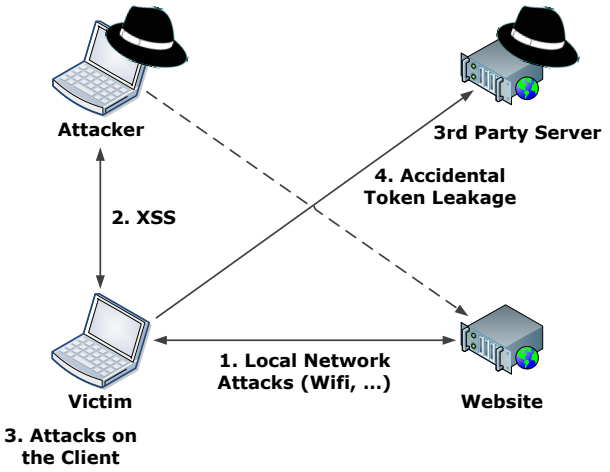

Figure 2. Attack Points for Session Hijacking

FaceNiff. In case the connection between victim and website is encrypted with HTTPS, the attacker might use sslstrip [25] or cookiemonster [32], as HTTPS as a sole countermeasure against session hijacking has been shown to often be insufficient. The token could also be obtained by an active attacker on the same network by means of ARP or DNS spoofing, redirecting the victim's communication in a man-in-the-middle attack, or DNS cache poisoning [18]. 2) If an attacker is able to inject Javascript into the website which is then executed at the victim side (XSS), he can transmit all necessary session tokens to himself and take over the session. 3) An attacker could access session tokens by attacking the browser directly using social engineering or a malicious browser extension, e.g., by tricking a victim into copy-pasting some Javascript into the URI bar of the browser. 4) In case of a poorly implemented Web application (HTTP referrer string), insecure transport (HTTP only) or network design (logging proxy server), an attacker might be able to access accidentally leaked tokens. This class of attacks would also include shoulder surfing (if the token is part of the URI) and improper usage of the Tor [6] anonymization network [27], [14].

B. Discussion

To counter the attacks listed above, SHPF relies on a combination of its features: the shared secret between the server and client using the SecureSession feature, and session hijacking detection using browser fingerprinting. An attacker would thus have to find out the secret, share the same IP and copy the behavior of the victim's browser - either by running the same browser version on the same operating system or by collecting the behavior of the browser over time.

The basic monitoring of HTTP information gives a baseline of protection. Binding a session to, e.g., an IP address makes it considerably harder for a remote attacker to attack, and a local attacker needs to be on the same local area network if the victim is behind NAT. Changes in the UserAgent or the HTTP header ordering are easily detectable, especially if careless attackers use sloppy methods for cloning header information, or only use some parts of the header for their user impersonation: Firesheep and FaceNiff, for example, both parse the header for session tokens instead of cloning the entire header. A recent manual analysis of the Alexa Top100 pages showed that only 8% of these very popular websites use basic monitoring in any form - notably eBay, Amazon and Apple [3]. Even though asynchronous challenges for fingerprinting on the attacker's machine could also simply be forwarded to the victim for the correct responses, the additional delay is detectable by the server.

We shall now discuss for each of the attacks outlined above how SHPF protects the session through active attack detection and prevention. Even though SHPF could work without HTTPS in certain configurations and environments, it should be used for starting the session, as without HTTPS bootstrapping the session becomes complicated, e.g., with respect to possible MITM attacks. As HTTPS is already used widely for user authentication, we assume that it is available at least for bootstrapping the SecureSession feature. SHPF has the following impact on the attack vectors: 1) Snooping or redirecting local network traffic can be detected at the server with either browser fingerprinting or using the shared secret, which is never transmitted in clear from SecureSession - both methods are equally suitable. 2) Cross-site scripting prevention relies on browser fingerprinting only, as the attacker could obtain session tokens by executing Javascript code in the victim's browser. The shared secret is not protected against such attacks. 3) Local attacks are also detected by browser fingerprinting only - the session secret is not safe, thus the attacker has to either run the same browser, or answer the asynchronous checks from the framework correctly. 4) Accidental token leakage is again prevented by both aspects, so even if the session is not encrypted by HTTPS the content is encrypted by the SecureSession feature and fingerprinting is used to detect changes in the used browser. Please see the original paper about SessionLock [1] for a detailed security analysis of the protocol.

SHPF does not intend to replace traditional security features for web sessions. While our approach cannot prevent session hijacking entirely it makes it considerably harder for the attacker. For sensitive websites with a high need for security, additional measures like 2-factor authentication or client-side certificates should be employed.

C. Limitations

Even though SHPF makes session hijacking harder, it has limitations: the HTTP headers and their ordering, as well as the UserAgent, are by no means security measures and can be set arbitrarily. However, if enough information specific to a browser is used in combination with ever

shorter update intervals for browsers, we believe that fingerprinting is suitable for preventing session hijacking. Secondly, SHPF does not protect against CSRF: An attacker who is able to execute code outside of the browser's sandbox, or has access to the hardware, can bypass our framework. Thus session hijacking is made harder in the arms race with the adversary, but not entirely prevented. Another limitation is the vulnerability to man-in-the-middle attacks: Diffie-Hellman in Javascript for shared secret negotiation is vulnerable to MITM, and either a secure bootstrapping process for session establishment or offline multifactor authentication is needed to protect the session against such adversaries.

D. Future Work

We plan to extend CSS fingerprinting with CSS selectors and CSS filters and intend to implement HTML5 fingerprinting as an additional SHPF feature. We furthermore plan to assess the tradeoff between the numbers of asynchronous challenges sent by the server to the total pool size of challenges for recent browser versions, as well as to measure the entropy of each fingerprinting method in practice. Even though the SHPF features for CSS (and soon HTML5 fingerprinting) are not designed to be used as single-use challenges within a session, we believe that measuring the entropy on a large set of users would be beneficial for the area of browser fingerprinting.

VI. Results

In general, the performance impact of running SHPF on the server is negligible as most of the processing is implemented as simple database lookups. Only a few kilobytes of RAM are needed per session and client for all features combined, while the overhead on the network is around 100 kilobytes (mostly for the libraries used by our framework - they need to be transferred only once due to browser caching). A mere 15 lines of code are needed to include SHPF in existing websites (see Appendix A), while the features each consist of a few hundred lines of code on average, with SecureSession being by far the biggest feature (about 4000 lines). Existing Web applications implement far more complicated logic flows and information processing capabilities then SHPF. [11]

The biggest impact on performance is caused by the generation of the primes for the Diffie-Hellman key exchange. We used a small but diverse set of devices to assess the clients' performance for creating the shared secret: a notebook (i7 CPU with 2 Ghz), a netbook (AMD Sempron with 1.5 Ghz) and two different smartphones (iPhone 4S and Nexus S). On the notebook, the latest versions of Chrome and Firefox at the time of writing (Chrome 18 and Firefox 12) were the fastest browsers for this operation, while Internet Explorer 9 was up to four times slower. As smartphones are limited with regard to CPU performance, they were even slower. A comparison of runtime needed for generating primes of different length can be seen in Figure 3. Depending on the security need of the website this overhead should be considered, as well as the amount of expected mobile users. The overhead caused by CSS fingerprinting on the client side is negligible compared to regular website rendering.

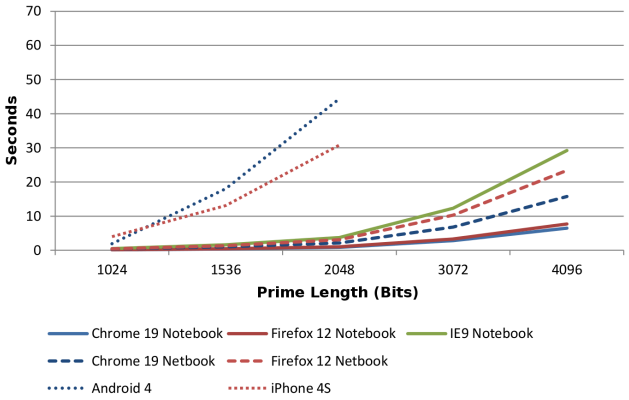

Figure 3. Performance of Prime Number Generation

A. Related Work

In the area of browser fingerprinting, different approaches have been used to identify a given browser. Panopticlick[12] relies on the feature combination of UserAgent string, screen resolution, installed plugins and more to generate a unique fingerprint [7] that allows the tracking of a given browser even if cookies are disabled. Even though the features of Panopticlick, such as screen resolution or installed browser plugins, are not yet fully incorporated in our framework, we are planning to do this in the near future. Other recent work in the area of browser fingerprinting identifies a client's browser and its version as well as the underlying operating system by fingerprinting the JavaScript engine [9]. While the approach in [28] uses various well-known JavaScript benchmarks to generate a unique fingerprint based on timing patterns, [35] employs a JavaScript conformance test to identify subtle differences in the conformance of the underlying JavaScript engine. Another recent method uses website rendering differences as presented in [29]. Like SHPF, these methods allow the detection of a modified or spoofed UserAgent string, as it is not possible to change the behavior of core browser components like the rendering or the JavaScript engine within a browser.

With regards to privacy, cookies and browser fingerprinting can be employed to track a user and their online activity. A survey on tracking methods in general can be found in [26]. Other work has recently shown that the UserAgent is often sufficient for tracking a user across multiple websites or sessions [43]. Intersection attacks on browsing history [31] or social networking

[11] We will release our datasets along with the source code once the paper is accepted.

[12] https://panopticlick.eff.org

sites [42] can be used to identify users. Session hijacking has been shown to allow access to sensitive information on social networking sites [15]. Finally, session hijacking is often conducted using cross-site scripting (XSS) attacks that are used to send the session information to an attacker. While this can be employed to protect a user from entering the password at an insecure terminal [3], it is often used maliciously, e.g., to impersonate the victim. Different approaches have been implemented to protect users from XSS on the client side [19], [39], [30] as well as on the server side [17].

The OWASP AppSensor project[13] is a framework that offers similar features as SHPF for Web applications: It can detect anomalies within a session and terminate it if necessary. However, it only uses a very limited set of checks compared to SHPF, namely the IP and the UserAgent string.

VII. Conclusion

In this paper, we presented our framework SHPF, which is able to raise the bar for session hijacking significantly. It detects and prevents attacks and hijacking attemps of various kinds, such as XSS or passive sniffing on the same network (Wi-Fi). We furthermore proposed two new browser fingerprinting methods based on HTML5 and CSS, which can identify a given browser. SHPF uses browser fingerprinting to detect session hijacking by constantly checking (e.g., with every request) if the browser is still behaving as it did when the session was started. SHPF can be configured to run with different security levels, allowing additional security checks for sensitive sessions or session parts. Future and upcoming fingerprinting methods can be incorporated easily.

Appendix

Appendix A - SHPF Example

```
include ('../SHPF/SHPF.php');

$shpf = new SHPF\SHPF ();
$shpf->setCheckFailedHandler ('\failedHandler');
$shpf->getOutput()->includeJSLibrary = false;

$serverEnvFeature = new SHPF\Features\HttpHeader\HttpHeaderFeature ($shpf);
$serverEnvFeature->setCheckAll (true);
$shpf->addFeature ($serverEnvFeature);

$shpf->addFeature (new SHPF\Features\SecureSession\SecureSessionFeature ($shpf));
$shpf->addFeature (new SHPF\Features\CSSFingerprint\CSSFingerprintFeature ($shpf));

$ret = $shpf->run ();


$output = Registry::get ('smarty');

$output->append ($shpf->getOutput()->flushHead(true), 'head');
$output->append ($shpf->getOutput()->flushHTML(true));
$output->append ($shpf->getOutput()->flushJS(true));
```

References

[1] B. Adida. Sessionlock: Securing web sessions against eavesdropping. In *Proceeding of the 17th International Conference on World Wide Web (WWW)*, pages 517–524. ACM, 2008.

[2] F. Ates and P. Irish. Modernizr - frond-end development done right. *Online at http://modernizr.com/*, 2006.

[3] E. Bursztein, C. Soman, D. Boneh, and J. Mitchell. Session-juggler: secure web login from an untrusted terminal using session hijacking. In *Proceedings of the 21st international conference on World Wide Web*, pages 321–330. ACM, 2012.

[4] E. Butler and I. Gallagher. Hey web 2.0: Start protecting user privacy instaed of pretending to. *ToorCon 2010*, 2010.

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

[6] R. Dingledine, N. Mathewson, and P. Syverson. Tor: the second-generation onion router. In *Proceedings of the 13th USENIX Security Symposium*, 2004.

[7] P. Eckersley. How unique is your web browser? In *Proceedings of Privacy Enhancing Technologies (PETS)*, pages 1–18. Springer, 2010.

[8] P. Eckersley and J. Burns. Is the ssliverse a safe place? *Talk at 27C3. Slides from https://www. eff. org/files/ccc2010. pdf [online*, 2011.

[9] E. ECMAScript, E. C. M. Association, et al. Ecmascript language specification. Online at http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf.

[10] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Rfc 2616: Hypertext transfer protocol–http/1.1, 1999. *Online at http://www.rfc.net/rfc2616.html*, 1999.

[11] D. Goodin. Tunisia plants country-wide keystroke logger on facebook. Online at http://www.theregister.co.uk/2011/01/25/tunisia_facebook_password_slurping/, retrieved 2012-11-19.

[12] N. Heninger, Z. Durumeric, E. Wustrow, and J. A. Halderman. Mining your Ps and Qs: Detection of widespread weak keys in network devices. In *Proceedings of the 21st USENIX Security Symposium*, Aug. 2012.

[13] R. Holz, L. Braun, N. Kammenhuber, and G. Carle. The ssl landscape: a thorough analysis of the x. 509 pki using active and passive measurements. In *Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference*, pages 427–444. ACM, 2011.

[14] M. Huber, M. Mulazzani, and E. Weippl. Tor http usage and information leakage. In *Communications and Multimedia Security*, pages 245–255. Springer, 2010.

[15] M. Huber, M. Mulazzani, and E. Weippl. Who On Earth Is Mr. Cypher? Automated Friend Injection Attacks on Social Networking Sites. In *Proceedings of the IFIP International Information Security Conference 2010: Security and Privacy (SEC)*, 2010.

[16] S. Ilinsky. Xmlhttprequest.js - cross-browser xmlhttprequest 1.0 object implementation, 2007. Online at http://www.ilinsky.com/articles/XMLHttpRequest/.

[17] M. Johns. Sessionsafe: Implementing xss immune session handling. *Computer Security–ESORICS 2006*, pages 444–460, 2006.

[13] https://www.owasp.org/index.php/OWASP_AppSensor_Project

[18] D. Kaminsky. Black ops 2008–it's the end of the cache as we know it. *Black Hat USA*, 2008.

[19] E. Kirda, C. Kruegel, G. Vigna, and N. Jovanovic. Noxes: a client-side solution for mitigating cross-site scripting attacks. In *Proceedings of the 2006 ACM symposium on Applied computing*, pages 330–337. ACM, 2006.

[20] M. Kirkpatrick. Ashton kutcher's twitter account hacked at ted. Online at http://www.readwriteweb.com/archives/ashton_kutchers_twitter_account_hacked_at_ted.php, retrieved 2012-07-19.

[21] A. Langley, N. Modadugu, and W.-T. Chang. Overclocking ssl. In *Velocity 2010*, 2010.

[22] N. Leavitt. Internet security under attack: The undermining of digital certificates. *Computer*, 44(12):17–20, 2011.

[23] A. Lenstra, J. Hughes, M. Augier, J. Bos, T. Kleinjung, and C. Wachter. Ron was wrong, whit is right. *IACR eprint archive*, 64, 2012.

[24] G. F. Lyon. *Nmap Network Scanning: The Official Nmap Project Guide To Network Discovery And Security Scanning*. Nmap Project, 2009.

[25] M. Marlinspike. sslstrip. Online at http://www.thoughtcrime.org/software/sslstrip/, retrieved 2012-07-26.

[26] J. Mayer and J. Mitchell. Third-party web tracking: Policy and technology. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 413–427. IEEE, 2012.

[27] D. McCoy, K. Bauer, D. Grunwald, T. Kohno, and D. Sicker. Shining light in dark places: Understanding the tor network. In *Privacy Enhancing Technologies*, pages 63–76. Springer, 2008.

[28] K. Mowery, D. Bogenreif, S. Yilek, and H. Shacham. Fingerprinting information in javascript implementations. In *Proceedings of Web 2.0 Security & Privacy Workshop (W2SP)*, 2011.

[29] K. Mowery and H. Shacham. Pixel perfect: Fingerprinting canvas in html5. In *Proceedings of Web 2.0 Security & Privacy Workshop (W2SP)*, 2012.

[30] N. Nikiforakis, W. Meert, Y. Younan, M. Johns, and W. Joosen. Sessionshield: lightweight protection against session hijacking. *Engineering Secure Software and Systems (ESSOS)*, pages 87–100, 2011.

[31] Ł. Olejnik, C. Castelluccia, and A. Janc. Why johnny can't browse in peace: On the uniqueness of web browsing history patterns. In *5th Workshop on Hot Topics in Privacy Enhancing Technologies (HotPETs)*, 2012.

[32] M. Perry. Cookiemonster: Cookie hijacking. *Online at http://fscked.org/projects/cookiemonster*, 2008.

[33] M. Pilgrim. Dive into html5, 2010.

[34] B. Ponurkiewicz. Faceniff. *Online at http://faceniff.ponury.net*, 2011.

[35] P. Reschl, M. Mulazzani, M. Huber, and E. Weippl. Poster abstract: Efficient browser identification with javascript engine fingerprinting. *Annual Computer Security Applications Conference (ACSAC)*, 12 2011.

[36] A. Sotirov, M. Stevens, J. Appelbaum, A. Lenstra, D. Molnar, D. Osvik, and B. de Weger. Md5 considered harmful today. *Creating a roque CA certificate*, 2008.

[37] D. Stuttard and M. Pinto. *The Web Application Hacker's Handbook: Finding and Exploiting Security Flaws*. Wiley, 2011.

[38] A. van Kesteren. Html 5 differences from html 4. *Working Draft, W3C*, 2008.

[39] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Cross-site scripting prevention with dynamic data tainting and static analysis. In *Proceeding of the Network and Distributed System Security Symposium (NDSS)*, volume 42, 2007.

[40] W3C. Webstorage, 2011. Online at http://www.w3.org/TR/webstorage/.

[41] w3schools. Html5 tag reference. *Online at http://www.w3schools.com/html5/html5_reference.asp*, 2012.

[42] G. Wondracek, T. Holz, E. Kirda, and C. Kruegel. A practical attack to de-anonymize social network users. In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 223–238. IEEE, 2010.

[43] T. Yen, Y. Xie, F. Yu, R. Yu, and M. Abadi. Host fingerprinting and tracking on the web: Privacy and security implications. In *Proceedings of the 19th Annual Network & Distributed System Security Symposium (NDSS)*. NDSS, 2012.

[44] S. Yilek, E. Rescorla, H. Shacham, B. Enright, and S. Savage. When private keys are public: results from the 2008 debian openssl vulnerability. In *Proceedings of the 9th ACM SIGCOMM conference on Internet measurement conference*, pages 15–27. ACM, 2009.